computer language

AN
AUTOINSTRUCTIONAL
INTRODUCTION
TO

fortran

McGRAW-HILL BOOK COMPANY, INC.

computer language

computer language

AN AUTOINSTRUCTIONAL INTRODUCTION TO fortran

HARRY L. COLMAN
Armour Research Foundation,
Illinois Institute of Technology

formerly with
Western Data Processing Center

CLARENCE SMALLWOOD, *Educational Consultant*
Western Data Processing Center
Graduate School of Business Administration
University of California, Los Angeles

Foreword by
GEORGE W. BROWN, *Director*
Western Data Processing Center

McGRAW-HILL BOOK COMPANY, INC.
New York *San Francisco* *Toronto* *London* **1962**

Copyright © 1962 by the McGraw-Hill
Book Company, Inc. Printed in the United States
of America. All rights reserved. This book,
or parts thereof, may not be reproduced
in any form without permission of the publishers.

Library of Congress Catalog Card Number: 62-17640

1234567890BB7210698765

Foreword

This autoinstructional text was originally designed to solve a difficult training problem at the Western Data Processing Center. The center serves academic research projects at the University of California, Los Angeles, and over seventy other institutions of higher learning in the thirteen Western states. Most users wish to use the Fortran Programming System even though they come from many different academic disciplines and their technical backgrounds vary markedly. The WDPC runs a Fortran Open Shop. Training problems at WDPC are similar to those met in most such open shops but exist here in exaggerated form. It would, for example, be extremely impractical to send instructors on monthly tours of seventy institutions. It is undesirable to have one person deliver the same lecture month after month, both because it is not sufficiently challenging for the instructor and because the audiences are so heterogeneous both in their interests and in their technical backgrounds.

Although the training problem at WDPC may be extreme, many of the relevant circumstances have counterparts at other academic and industrial installations. This text constitutes a self-contained training system for introducing students of almost any background or professional interest to the details of Fortran coding and the art of computer programming.

The system provides detailed information for students who have little or no familiarity with algorithmic notation but also permits students with this familiarity to skim that information efficiently. The examples and exercises are based on commonplace experience and do not require knowledge of a specialized application. Furthermore, the system contains an indexed reference manual that is cross-referenced to the program for periodic, voluntary review. Upon completion of the program, students will have written their first Fortran program. This program can be keypunched and compiled and executed. A short period of advanced training on the special features of any particular system should produce Fortran programmers ready to write useful computer programs for that system.

GEORGE W. BROWN
Director
Western Data Processing Center
University of California, Los Angeles

Preface

Fortran is the name of a simple algebra-like language for programming computers. It is a contraction of "FORmula TRANslator" and is potentially valuable for solving any problem that can be expressed as a series of numerical relations or algebraic formulas.

Such problems are common in most professional and academic specialties. People working in these specialties are rapidly coming to depend on computers to alleviate the cost, the drudgery, and the often insurmountable hurdles of manual and mechanical calculations. The computer, however, has become not only a symbol of relief from tedium, but an indispensable tool for the day-to-day operation of large-scale data processing systems, for the conduct of research projects, and for the solutions of complex, practical problems. As this trend continues, it becomes essential for more and more people to acquaint themselves with computer programming.

Fortran is a relatively "old" computing language. The first version of it, for the IBM 704, was distributed in 1957. Since then it has been expanded and enriched and is currently available for programming many manufacturers' computers. The Fortran presented in this text is taken from IBM FORTRAN II. To avoid an unnecessary source of potential confusion, however, some of the nonessential, optional flexibility has been omitted. What remains is fairly universal and basic. Once this basic information has been learned and applied, the additional features of a particular Fortran system can be learned with relative ease as the need arises.

This book is an introduction, not only to Fortran, but also to the art of computer programming. It will be helpful to those who are planning to use the computer for the first time and to those who wish merely to acquaint themselves with the rudiments in Fortran programming. It should be useful to managers and administrators who need a basic acquaintance with the concepts and terminology of computer programming and to high school and college students in mathematics, statistics, engineering, and business administration.

The teaching technique used is called *autoinstructional.* It is based on B. F. Skinner's reinforcement theory of learning—the theory of programmed learning and teaching machines. Traditional applications of this theory require students to respond overtly to *every* new unit of information by writing down a word, numbers, or phrase, the correctness of which is *immediately* verified in the text. Experiments, particularly by Arnold Roe at UCLA, indicate, however, that under some circumstances, if the overt response and verification are omitted, the result is a program of instruction that is equally as effective and significantly more efficient. This mode of autoinstruction is usually called the *no-response* mode. It has been used extensively and, we believe, successfully in the present text.

The book also differs from traditional autoinstructional texts in the layout of the pages. Each page was individually designed by Elizabeth Paine, on the assumption that its unique appearance will aid those who remember information visually.

The no-response sections are supplemented by exercises and program-writing tasks. The exercises were added, not as instructional, but as a means of building confidence. The tasks are larger segments of behavior, and are analysed and corrected upon their completion, as part of the autoinstructional technique.

The tasks, the exercises, and the examples were all chosen for the generality of their content. This was also done so that students without particular background knowledge would not be aversively affected.

The program has never been experimental. It was designed as a practical replacement for eight hours of traditional instruction. Early drafts underwent informal trials, which resulted in extensive revisions and retrials. We hope the end product will add a modicum of efficiency to the job of learning Fortran.

We thank the following institutions and people: the Western Management Science Institute, Graduate School of Business Administration, University of California, Los Angeles, for its support, under a grant from the Ford Foundation, of the preparation and reproduction of the early drafts of the program; the director and staff of the Western Data Processing Center for their guidance and assistance: Mr. Arnold Roe of the UCLA engineering department's Teaching Systems Project for consultation; and the students who exposed themselves to the early drafts and provided feedback so essential for effective revision.

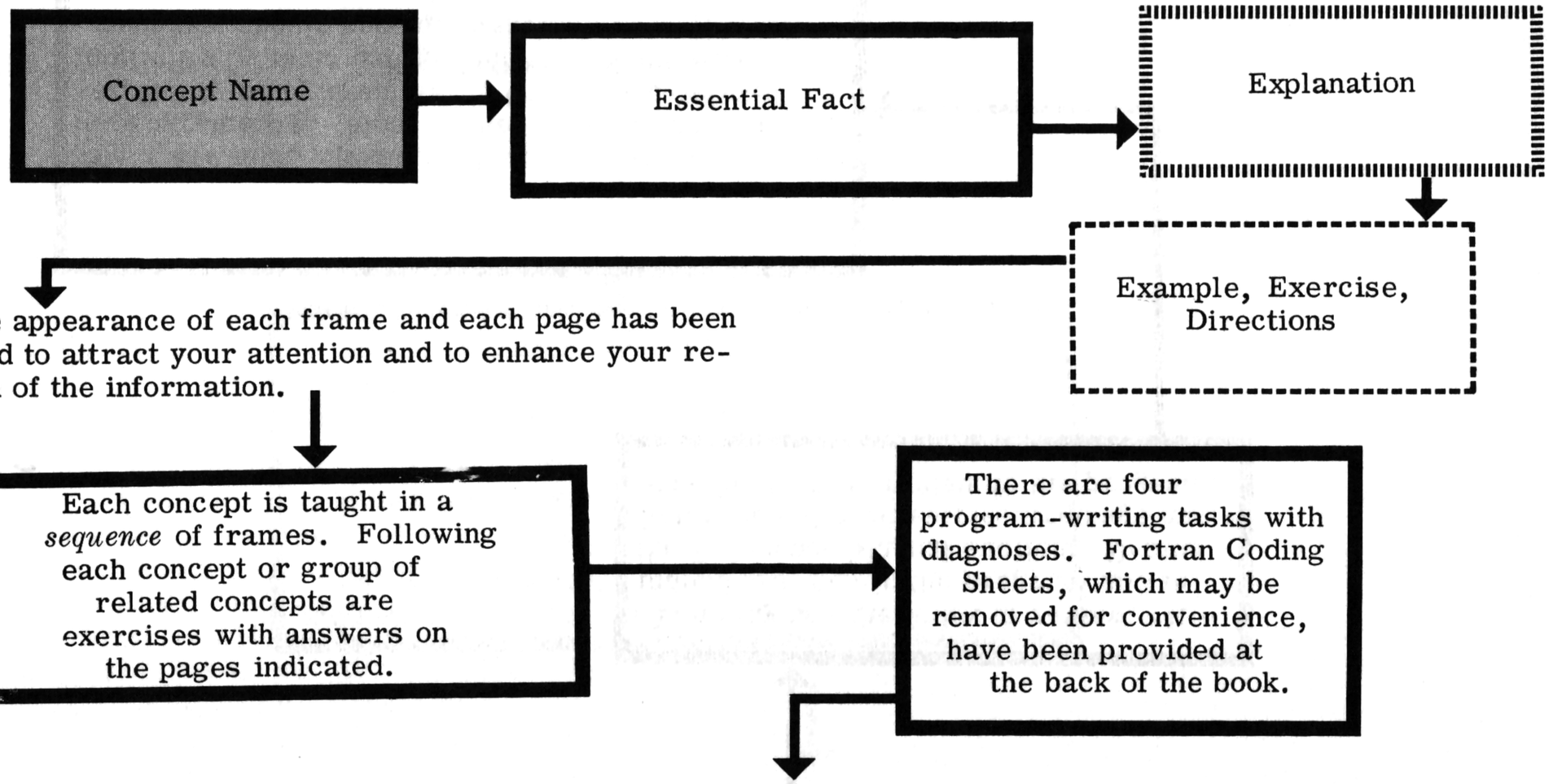
HARRY L. COLMAN

CLARENCE SMALLWOOD

To the Reader

This text is an autoinstructional program. It has been written and designed to provide an efficient, self-contained system for learning the basic concepts of Fortran programming.

The subject matter is presented in short, concise increments printed inside a frame. The type of frame denotes the kind of information inside it.



You may use the Index to review. It indicates pages in the program and in the Reference Manual section. Beside Reference Manual entries are page numbers for further review in the program.

There are many specialized words in the vocabulary of programming. Most of these words are defined explicitly or by example. Others are defined implicitly by their use in contexts where their meanings should become clear.

The program assumes that you have no special background in computer programming. Therefore, proceed rapidly in those parts which seem clear, and read more carefully in other parts.

Learning time has varied from two to over ten hours.

Use this autoinstructional system in the way that suits you best.

✶ If the subject matter is easy to understand, concentrate on the Essential Facts, and skim over the Explanations and Examples.

☆ If, while working the Exercises, you are confident your answer is correct, omit verifying it.

✶ Review concepts at any time, using either the Index or the Table of Contents.

☆ Make notes on questions that are not answered in the program. If they are not answered in subsequent sections, seek help from a more advanced Fortran reference manual or from an experienced Fortran programmer.

★ If you are already familiar with the concepts covered in Part One: Introduction, start with Part Two: Program Structure.

Table of Contents

Foreword v
Preface vii
To the Reader ix

Part One: INTRODUCTION

Computer Programs 1 Computer Instructions 3 Flow Diagrams 6 Memory 17 Object Programs and Source Programs 20 Fortran Source Program Compilation 23 Kinds of Fortran Statements 27 Fortran Coding Sheets 29 Punched Cards 34 Exercises 39

Part Two: PROGRAM STRUCTURE

A Job 43 Subroutines 44 Main Program 47 Interprogram Flow of Control 48 Exercises 53 Main Program Task 54 Diagnosis 55 Comment Statements 57

Part Three: VARIABLES AND CONSTANTS

Fortran Variables 58 Exercises 63 Fortran Constants 63 Exercises 65 Subscripted Variables 66 DIMENSION Statements 67 Exercises 69 Common Storage 70 Exercises 73

Part Four: INPUT STATEMENTS

Input Data Fields 75 Input FORMAT Description 77 Multiple Fields 80 READ Statements 82 Relations between Fields, Descriptions, and the Variable List 84 Exercises 87 Input Task and Diagnosis 89

Part Five: ARITHMETIC EXPRESSIONS

Expressions 94 The Hierarchy of Operations 95 Parentheses 97 Exercises 100 Mixed Expressions 103 Exercises 105

Part Six: ARITHMETIC STATEMENTS

General Form 106 Incrementing 107 Accumulating 108 Defining a Variable 108 Conversion 109 Exercises 111

Part Seven: CONTROL STATEMENTS

Normal Flow of Control 112 Unconditional Transfer of Control 114 Conditional Transfer of Control 115 IF-Loops 117 DO-Loops 120 Exercises 126 A Sorting Example 130 Sorting Task 139 Diagnosis 141

Part Eight: OUTPUT STATEMENTS

Relation to Input 147 Output Field Specifications 148 Carriage Control 152 Relations between Fields, Specifications, and the Variable List 155 Exercises 156 Output Task 158 Diagnosis 159 Conclusion 161

Answers to Exercises 162
Reference Manual 179
Index 195
Fortran Coding Sheets 197

Part One: INTRODUCTION

Computer Programs ● Computer Instructions Flow Diagrams ● Memory ● Object Programs and Source Programs ● Fortran Source Program Compilation ● Kinds of Fortran Statements Fortran Coding Sheets ● Punched Cards ● Exercises

Computer Programs

A program for a computer is like a set of instructions for a clerk who is running a desk calculator.

We can tell the calculator clerk *what* to do with some numbers without saying what the particular numbers are.

We can, for instance, instruct the clerk to add all the numbers on List A, write that total on a piece of blue paper, and label it Total of List A.

The clerk will know what to do without knowing the numbers on the list or the purpose for adding them.

In fact, the same instructions to the clerk will work for *any* list of numbers that we choose to call "List A."

On one day List A may contain one set of numbers, and on another day another set.

Computer Programs are like the set of instructions to the calculator clerk.

Notice that there are three fairly distinct operations involved:

Input - e. g., List A

Calculation - e. g., Accumulating the total

Output - e. g., Writing the total on a piece of blue paper.

All three phases are involved in most computer programs. The instructions that make up a computer program cause the computer to read the input data, perform some calculations, and print the results.

Naturally, the analogy between the calculator clerk and the computer has limits.

We can tell the clerk to add the numbers without saying how many there are, and without saying, "Now add the next one, now the next one, etc."

We must, however, instruct the computer first to add the first number, then to add the second number, then the third, etc.

Computer Instructions

There is no computer instruction such as

"Calculate the Mean."

To cause a computer to calculate a mean, we must order it to perform a large number of distinct operations.

There is no computer instruction like

"Simulate an Oil Refinery."

The computer program that would simulate an oil refinery would be comprised of many thousand instructions related in a very complex way.

Planning and designing computer programs can be a very complicated affair. But nearly all programs require planning.

It would be marvelous, indeed, if the raw data of a problem could simply be "fed to the computer," and the computer could somehow analyze the problem, arrive at the correct solution, and carry out that solution to produce the desired results.

There is no computer that can do this.

We cannot give the computer *vague* instructions about what we want it to do.

We cannot, for example, order the computer to "compute the statistical significance" or to "simulate the average man," without adding certain pertinent details to the general instructions.

A computer is wired to react to a specific, limited set of machine instructions in a specific, prescribed way.

Our instructions to it must ultimately be a program of these machine instructions.

It is this fact that requires us to analyze and organize our problems and solutions to an extent that other methods often do not demand.

Whereas human beings have intuitive capabilities, the high-speed digital computer *does not.*

It is not possible to give a computer a vague or general instruction like

"Add any hundred numbers."

It is possible to give a computer *foolish* but precise (specific) instructions like

"Add every other number in the first hundred storage locations in memory."

The first instruction, the one the computer is *not capable* of acting on, is analogous to giving a child a ten-dollar bill and telling (instructing or programming) him to go to the supermarket and buy "food."

The second is analogous to foolishly adding to the original instructions, the more specific instruction to bring home from the market each second item of food of the first hundred articles he sees in the market.

Whereas the child may accurately deduce what we want from an instruction as vague as the first one, the computer cannot because it does not have the capability of understanding, comprehending, or interpreting, and therefore cannot deduce.

It cannot do this even though it has had years of previous experience with perfect programs.

The programmer must *clearly* specify, via precise instructions, exactly *what* he wants the computer to do and the order in which it is to be done.

Flow Diagrams

NOTE

The objective of this section is to acquaint you with Flow Diagramming.

The emphasis will be on the *order* of events in a program, not on the events themselves. We, therefore, advise you *not to be too concerned* with the contents of the diagram boxes in the illustrations.

Most beginners design and plan a computer program by drawing a Flow Diagram for the program.

A Flow Diagram is a graphic representation of *all* the sequences and alternate sequences of operations that are to be specified in a computer program.

A Flow Diagram shows the temporal and causal relationships between operations and decisions in a program.

In showing alternate paths in the sequence, a Flow Diagram must specify the conditions under which each and every alternate path could be followed.

The sequence of events is crucial. We must, for example, instruct the computer to input List A *before* we instruct it to add the first number on that list.

When computing a mean, we must instruct the computer to accumulate the total before dividing by the number of numbers.

Because Flow Diagrams are not themselves composed of computer instructions, there are no stringent rules for making them. You may use any notation you please.

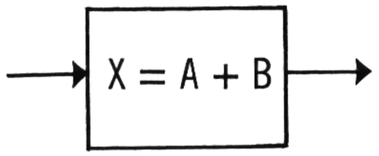
On the other hand, because the Flow Diagram will be used to write a computer program, it does no harm to orient it to computer operations.

There are many conventional techniques for drawing Flow Diagrams. Their use is recommended for the simple reason that other people will be able to understand them without learning a new set of techniques every time they look at a Flow Diagram.

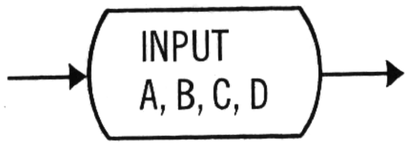
Conventional Flow Diagrams are composed of geometrical shapes connected by arrows.

The shapes are called "boxes" even though they are not all rectangular.

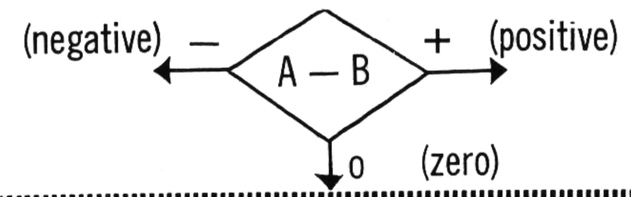
Rectangles are used to indicate operations, such as addition, subtraction, multiplication, division, and exponentiation.

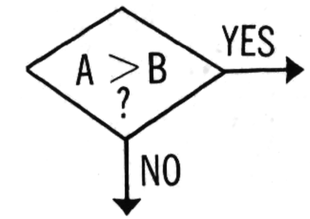


Rectangles with rounded ends are used to specify input and output operations.



Diamonds are used to specify branching decisions. The exit taken depends on the value of the expression inside the box.





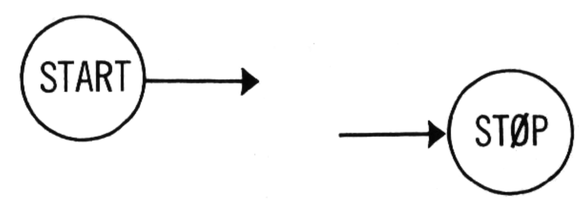
Explanation

If the result of the subtraction of B from A is negative, the minus exit is taken.

If the result of the subtraction of B from A is positive, the plus exit is taken.

If the result of the subtraction of B from A is zero, the zero exit is taken.

Large Circles are used to indicate starting and stopping points.



Arrows indicate the sequence in which the operations in the boxes are to be performed.

They specify what is called the

"Flow of Execution"

or the

"Flow of Control"

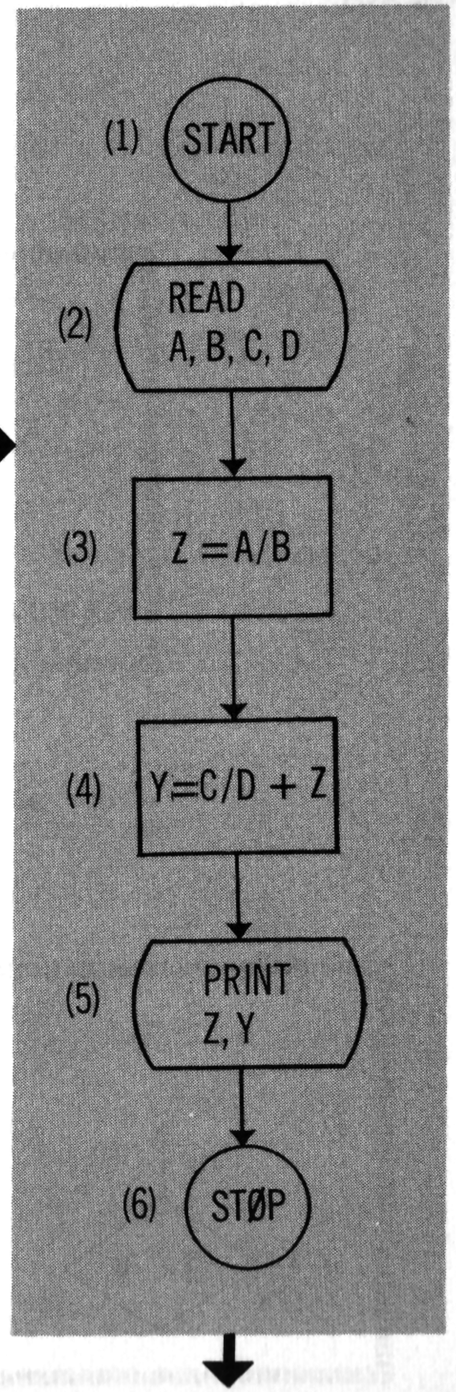
Inside the boxes, any notation may be used:

Algebra, English imperative sentences, Boolean symbology, Proofreader's marks, Swahili, Roman numerals, or cartoons.

However, because Fortran operations are algebraically oriented, you should use algebra wherever possible.

This is a Flow Diagram of a very simple, straightforward program. It *reads* (inputs) four values, *computes* two new values, and *prints* out the two new ones.

The numbers in parentheses opposite the boxes correspond to the explanatory frames on the next page.



READ THESE BOXES IN NUMERICAL ORDER

1 This is the starting point.

2 Four numbers are read in and designated

A, B, C, and D

3 Z is set equal to the result of the division of

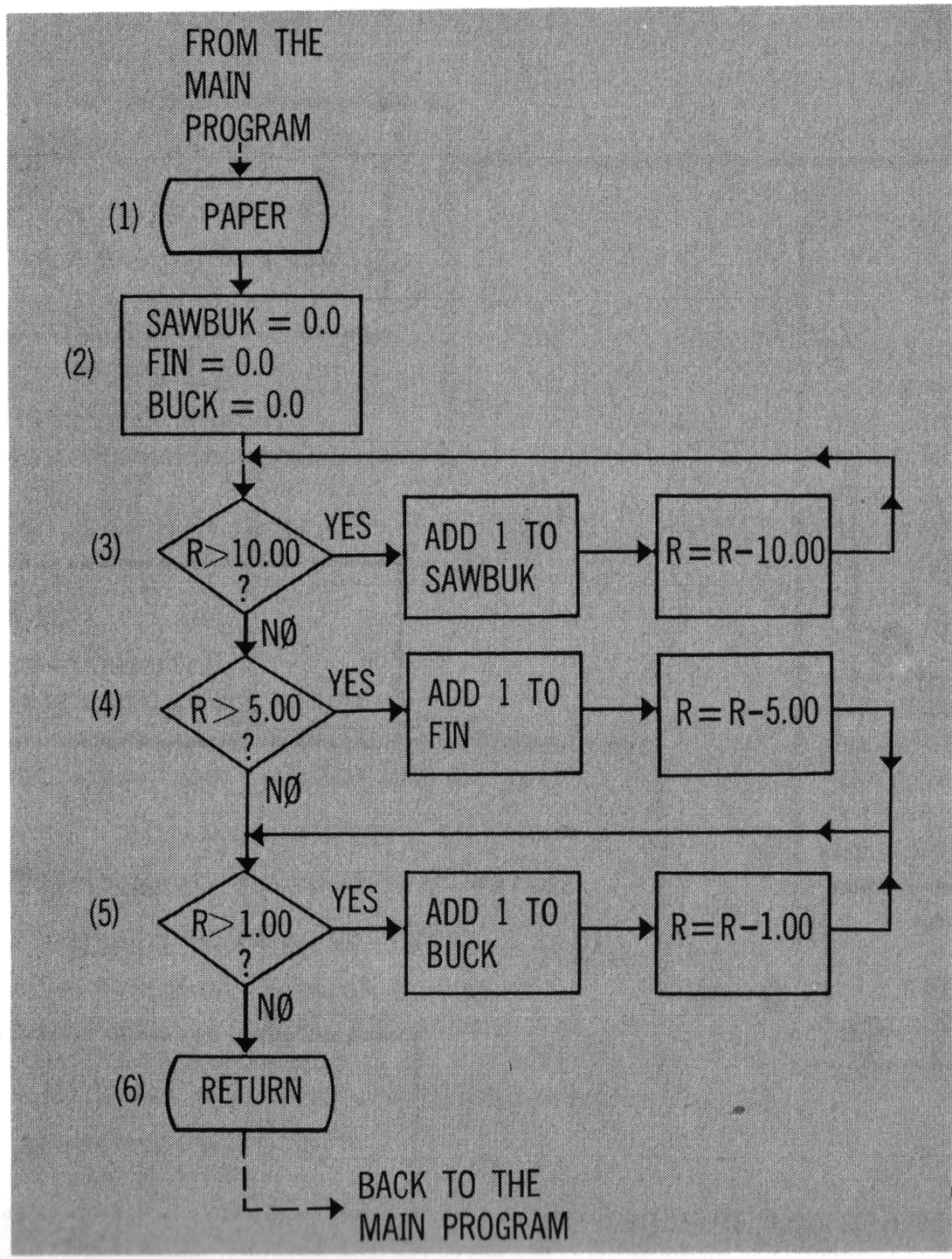
A by B

4 Y is set equal to the result of

C divided by D, plus Z

5 Z and Y are printed out.

6 This is the end of the program.



This is a Flow Diagram of a subroutine called PAPER.
The purpose of PAPER is to keep a tally of how many ten-five-and one-dollar bills will be needed in making change equal to "R."

(Assume the "R" is computed in some other subroutine.)

READ THESE BOXES IN NUMERICAL ORDER

1 This is the starting point of

PAPER

2 The counters, SAWBUK, FIN, and BUCK must all be started at zero.

3 If R, the value of change still to be tallied, is greater than $10.00,

add one to SAWBUK, and

Subtract 10.00 from R.

4 As soon as R is less than $10.00, decide if R is greater than $5.00.

If it is,

add one to FIN, and subtract 5.00 from R.

5 If R is greater than $1.00, add one to BUCK,

and subtract 1.00 from R.

Repeat this decision and operation until R is less than $1.00.

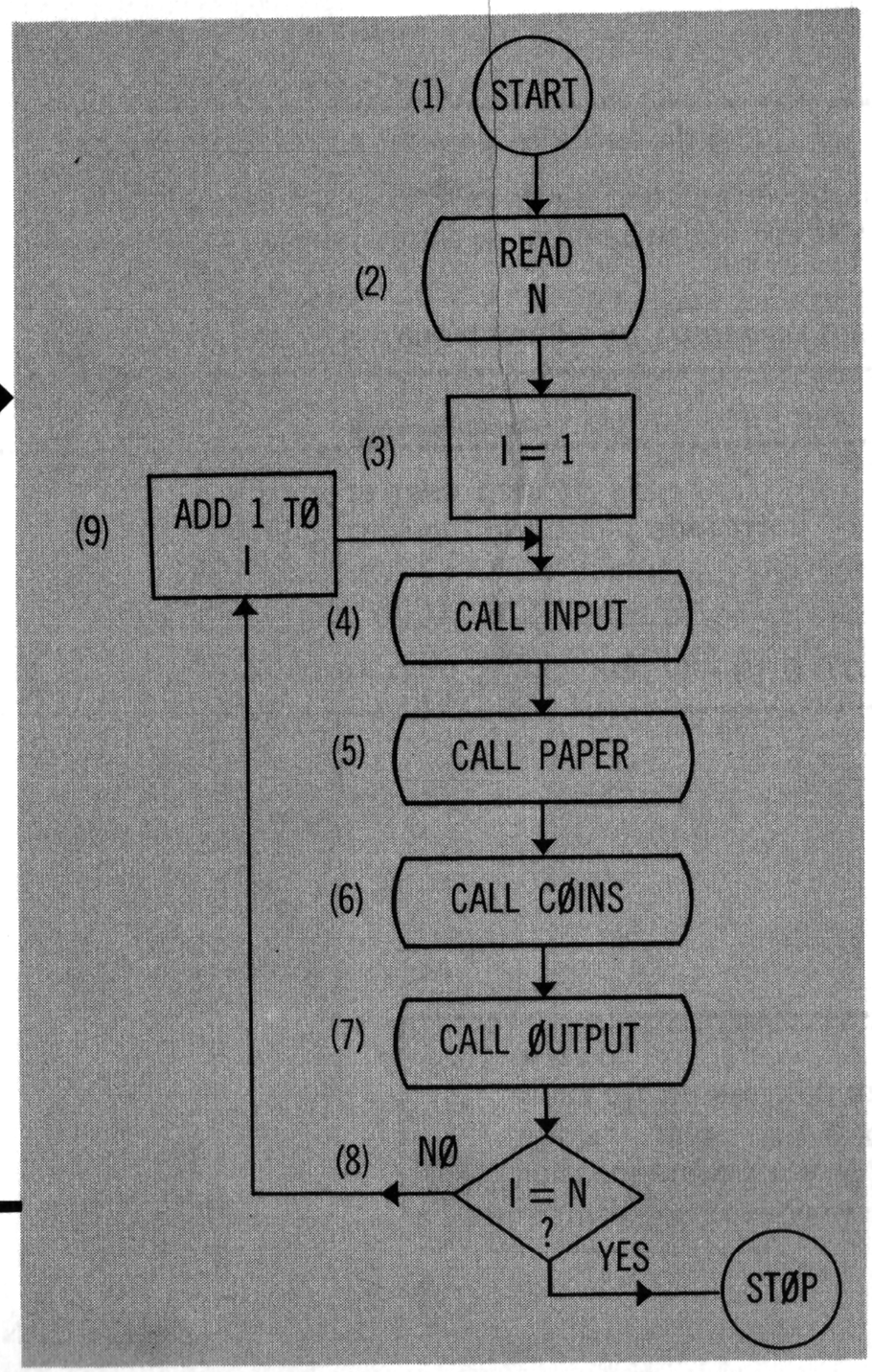
6 When R is less than $1.00 return to the Main Program.

(The next thing to happen in the Main Program will be to call another subroutine to compute

the coins needed to make up R.)

This is a Flow Diagram of a Main Program that calls four subroutines.

The purpose is to compute the number of bills and coins needed to make up change.



READ THESE BOXES IN NUMERICAL ORDER

1 This is the starting point

2 The value read in for N will be the total number of times change shall be computed.

3 I will be an index or counter for keeping track of the number of times change has been computed.

I will be incremented each time change has been computed.

4 CALL INPUT means that the INPUT subroutine shall operate at this point.

The RETURN statement in the INPUT subroutine will bring the flow to the exit arrow from this box.

INPUT reads in a transaction and computes a value for R, the amount of change needed.

5 CALL PAPER means that the PAPER subroutine shall operate at this point.

PAPER will compute the number of bills needed to make up R.

The RETURN statement in PAPER will return the flow to the exit arrow from this box.

6

CALL COINS means that the COINS subroutine will function at this point.

COINS will compute the amounts of nickels, pennies, dimes, etc., needed to make up R.

The RETURN statement in COINS will return the flow to the exit arrow from this box.

7

CALL OUTPUT means that the OUTPUT subroutine shall operate at this point.

OUTPUT will print out the numbers of each denomination of coin and bill needed to make up an amount of money equal to R.

The RETURN statement in OUTPUT will return the flow to the exit arrow from this box.

8

This is a decision box. If I is now equal to N, the program should terminate. This is the "yes" exit from the decision box.

If I does not yet equal N, the process must be repeated. Another cycle of change-making should begin.

9

Before the next cycle begins, I should be increased by one.

The first time through the cycle, I was one.

The second time through, I was two.

The last time through it will equal N.

10

This is the end of the Program.

Memory

The three basic parts of a computer are:

- The Input-Output Units
- The Memory Unit
- The Central Processing Unit (CPU)

Memory consists of several thousand "storage locations" or "memory locations." The contents of a storage location are called a "word."

Each storage location has a numerical "address." When programs are being executed, they always refer to storage locations (and the words in them) by using this numerical address.

There are approximately 32,000 storage locations in the IBM 709 and 7090.

A memory of this size is referred to as a 32K memory.

Each word is 36 binary digits—36 bits. Every pattern of 36 bits in a word represents a numerical value in the binary number system.

In fact, all machine instructions for the computer are words, and may be represented as patterns of 36 bits.

Words in memory may represent:

- Decimal numerical values converted to binary
- Octal numerical values converted to binary
- Binary numbers
- Binary-coded alphabetic information
- Machine instructions

When words are transmitted *from* a memory location, they are *not erased* in that memory location. A replica of the pattern of bits is transmitted.

However, when a bit pattern (a word) is stored in a location, it replaces and erases the word that was in that location before it.

Words are transmitted between the computer's memory and Central Processing Unit (CPU) in microseconds (millionths of a second).

This transmitting of words, of course, is always done in response to machine instructions in a program.

And this program of machine instructions is itself stored in the computer's memory.

A program of machine instructions is called an Object Program

Object Programs and Source Programs

A machine instruction is a pattern of binary digits in a computer word.

Machine instructions are transmitted in sequence to the Central Processing Unit (CPU).

The CPU is designed to react in a particular way to particular instructions.

A typical machine instruction causes the CPU to "clear" its accumulator "and add" to it the contents of the storage location specified in the instruction.

For example- the following binary machine instruction

00010100000000000000000000000010100

would cause the CPU to set its accumulator at zero and then to add to it the contents of storage location 010100.

A sequence of binary machine instructions, designed to perform some meaningful computations, is called an

Object Program

Fortunately, these days, it is no longer necessary to write programs in binary machine instructions.

Most professional programmers, and all amateurs, write programs in one or more of the many symbolic languages that have been developed for the purpose.

Programs written in these languages are called:

Source Language Programs

Associated with each Source Language Program is an elaborate system of computer programs that can translate Source Programs into

Object Programs.

In general, there are two levels of Source Program Languages:

1. One-to-one Languages
2. One-to-many Languages

In a one-to-one Source Language, there is a translation from *one* symbol in the Source Language to *one* machine instruction.

The translation from symbols to machine instructions is called "assembly." The translator is called the "assembler."

In a one-to-many Source Language there may be many machine instructions for each statement in the Source Language.

The translation is called "compilation," and the translator program is called the "Compiler."

Fortran Source Program Compilation

Fortran is a one-to-many Source Program Language. Statements in Fortran are compiled into Object Program machine instructions.

Programs written in Fortran are called Fortran Source Programs

The *computational* statements in Fortran resemble ordinary algebra.

Control and *input-output* statements resemble ordinary English language imperative sentences.

Storage statements resemble declarative sentences.

The transformation of a Fortran Source Program into an Object Program is called Compilation.

Compilation is done in the computer by a program called the Fortran Compiler

The Fortran Compiler Program analyzes each statement in the Fortran Source Program, and generates a sequence of machine instructions which, when executed, will perform precisely those computations, etc., specified in the Source Program.

Having successfully generated an Object Program, the compiler does two things with it:

1. A copy of the Object Program is temporarily stored on a special tape for subsequent execution.
2. Another copy is punched into tabulating cards for storage, and for execution at later dates.

As the compiler analyzes each statement, it uncovers certain kinds of errors. These are referred to as

Diagnostic Errors

The compiler can diagnose the following kinds of errors in a Source Program:

- Illegal characters
- Illegal combinations of characters
- Improper flow of control through the program.

A Source Program containing an error that the compiler can diagnose will not be compiled into an Object Program.

In the case of diagnosable errors, the compiler prints out the erroneous statement, and attempts to catalogue the kind of error.

The Fortran compiler cannot diagnose logical errors. It cannot deduce your intentions. It cannot compensate for even the most obvious mistakes in specifying computations, etc.

For example, if your program were computing areas of circles, and

you wrote 3.7416
instead of 3.1416

the compiler could not diagnose the error.

Programs containing errors that the compiler cannot diagnose will be successfully compiled. The Object Program so compiled, however, will certainly not produce the results intended by the author of the Source Program.

Therefore, it is essential that you be precise in writing a program, by paying close attention to even the smallest detail.

Kinds of Fortran Statements

A sequence of Fortran Statements comprises a Fortran Source Program.

There are two kinds of Source Program statements:

- Executable
- Nonexecutable

Executable statements result in machine instructions.

Nonexecutable statements tell the compiler what kind of storage requirements will be needed and how input and output data are laid out.

Nonexecutable statements affect the *design* of the Object Program. Executable statements affect the *content*.

There are four kinds of nonexecutable statements:

- Storage statements
- Format statements
- Subroutine-defining statements
- Comment statements

There are four kinds of executable statements:

- ★Arithmetic statements
- ☆Control statements
- ★Input statements
- ☆Output statements

The rules for writing these different kinds of statements will be discussed in the next section.

Fortran Coding Sheets

Statements in a Fortran Source Program are first written on coding sheets.

A typical short Main Program with a typical short subroutine looks something like the following:

✶ Note: Do not ponder the meanings of the Fortran statements in the samples.

They are not important at this point, and will become clear as you progress through the program.

FORTRAN CODING FORM

Program MAIN PRØGRAM
Coded By JIM BRØWN
Checked By SAM HAND

Identification (73–80)

Date TØDAY
Page 1 of 2

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–72)
C		THIS IS A MAIN PRØGRAM
C		IT READS VALUES FØR N AND FØR X(I), I = 1, N
C		
		DIMENSIØN X(999)
		CØMMØN X, N, AVERAGE
21		FØRMAT (I3)
		READ 21, N
22		FØRMAT (7F10.4)
		READ 22, (X(I), I = 1, N)
		CALL MEAN
		CALL EXIT
		END

FORTRAN CODING FORM

Program SUBRØUTINE MEAN
Coded By JIM BRØWN
Checked By SAM CLARK

Date TØDAY
Page 2 of 2

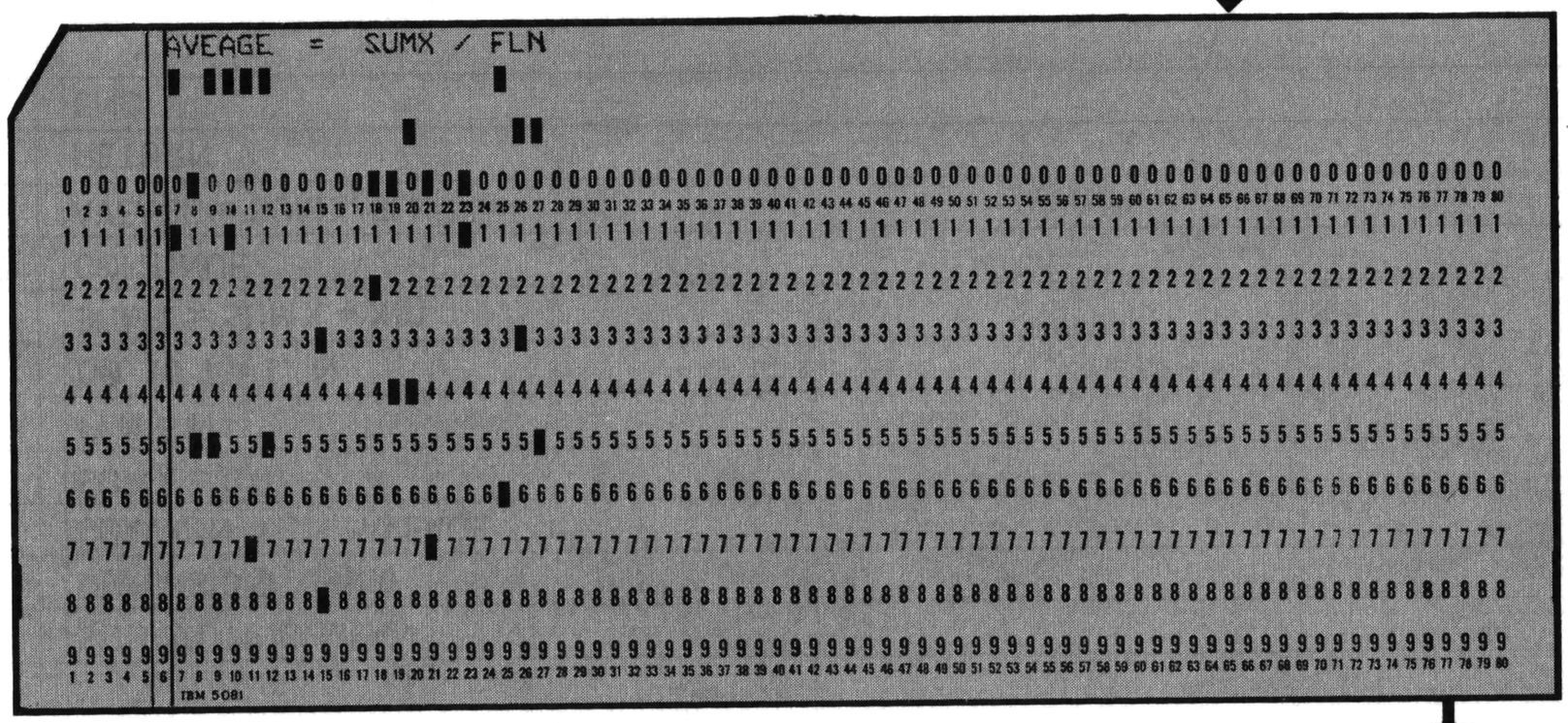
Identification (73–80)

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–72)
		SUBRØUTINE MEAN
		DIMENSIØN X(999)
		CØMMØN X, N, AVEAGE
		SUMX = 0.0
		FLN = N
		DØ 10 I = 1, N
		SUMX = SUMX + X(I)
10		CØNTINUE
		AVEAGE = SUMX/FLN
		RETURN
		END

Punched Cards

The information on each line of the coding sheet is punched on a separate tabulating card.



There are 80 columns in a tabulating card.

Columns 1 - 5 are used for the statement number.

(It is necessary to number some statements so that they may be referred to in other statements.)

Columns 7 - 72 are used for the body of the Fortran Statement.

Column 73 - 80 are used only for sequencing and identification.

Column 6 is normally left blank.

However, if a Fortran statement requires more than one line (one card), it may be continued on another one. In this case, the continuation card must have some character punched in column 6.

This character may be any character other than a blank.

To avoid confusion, the following printing conventions are recommended on Fortran Coding Sheets:

1 - One	I - Eye
2 - Two	Z - Zee
0 - Zero	Ø - Oh

The contents of Fortran statements are punched on tabulating cards by using a keypunch machine. This machine has a keyboard like a typewriter. Hitting a key causes a pattern of holes to be punched in a card and the character to be printed.

What is a Punched Card?

A punched card or tabulating card is a card of uniform size and shape that has holes punched in it according to an established code-pattern.

Punched cards have the important advantage of being relatively permanent, and they can be read quickly and accurately by several different kinds of machines. They are often called Hollerith Cards.

Fortran programs and data are punched into cards using a code-pattern:

Each card consists of 80 vertical columns.

Each column consists of 12 possible punch positions.

Each column is capable of containing the punched code for *one* character.

The code-pattern for the character is punched in the column, and the character itself is usually printed on top at the head of the column.

Printing the character at the top of the column alleviates the necessity of memorizing the code-pattern.

The printed character is *never* read by the machine; it is only for human convenience.

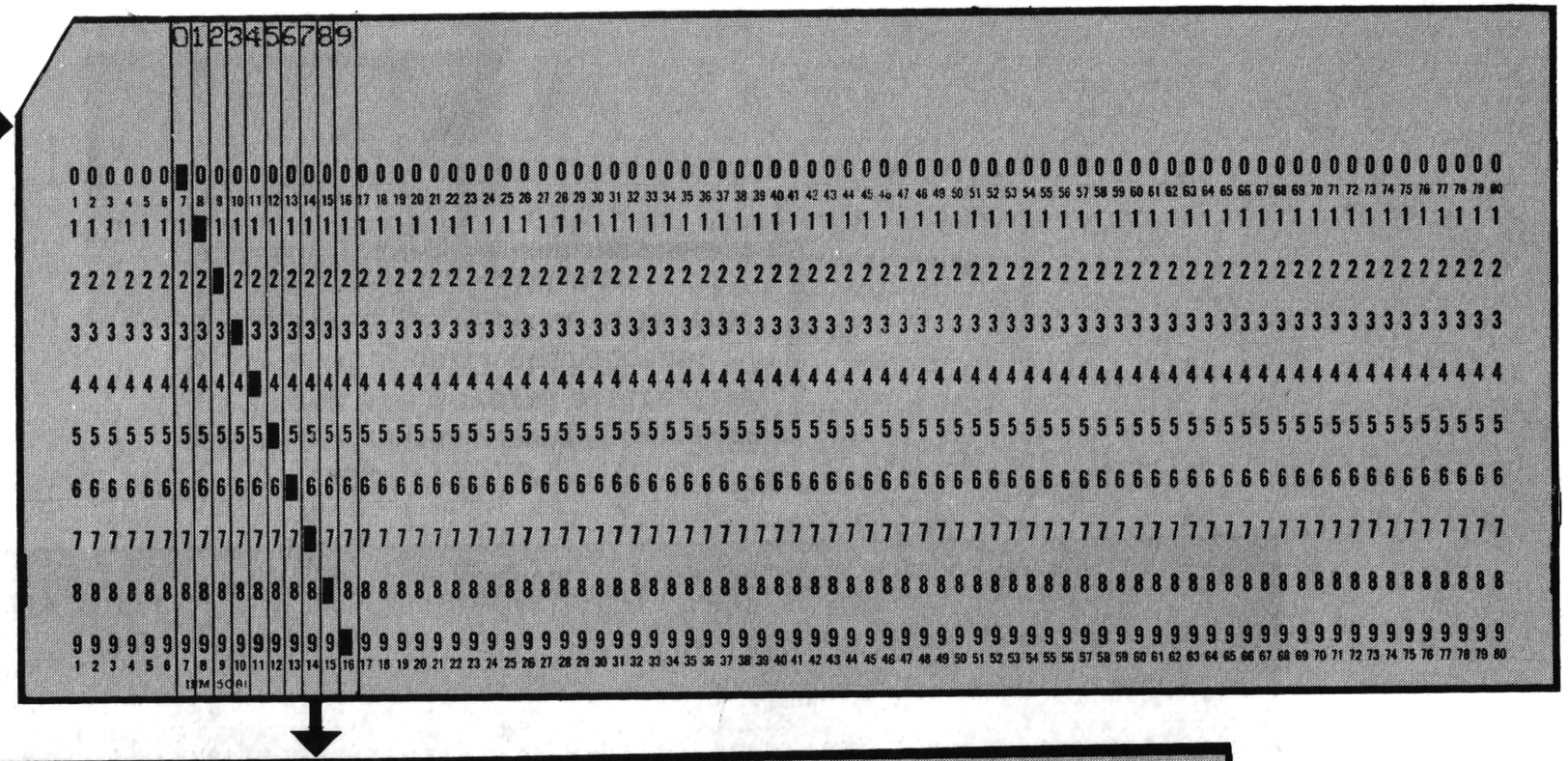
Following are Sample Cards with the

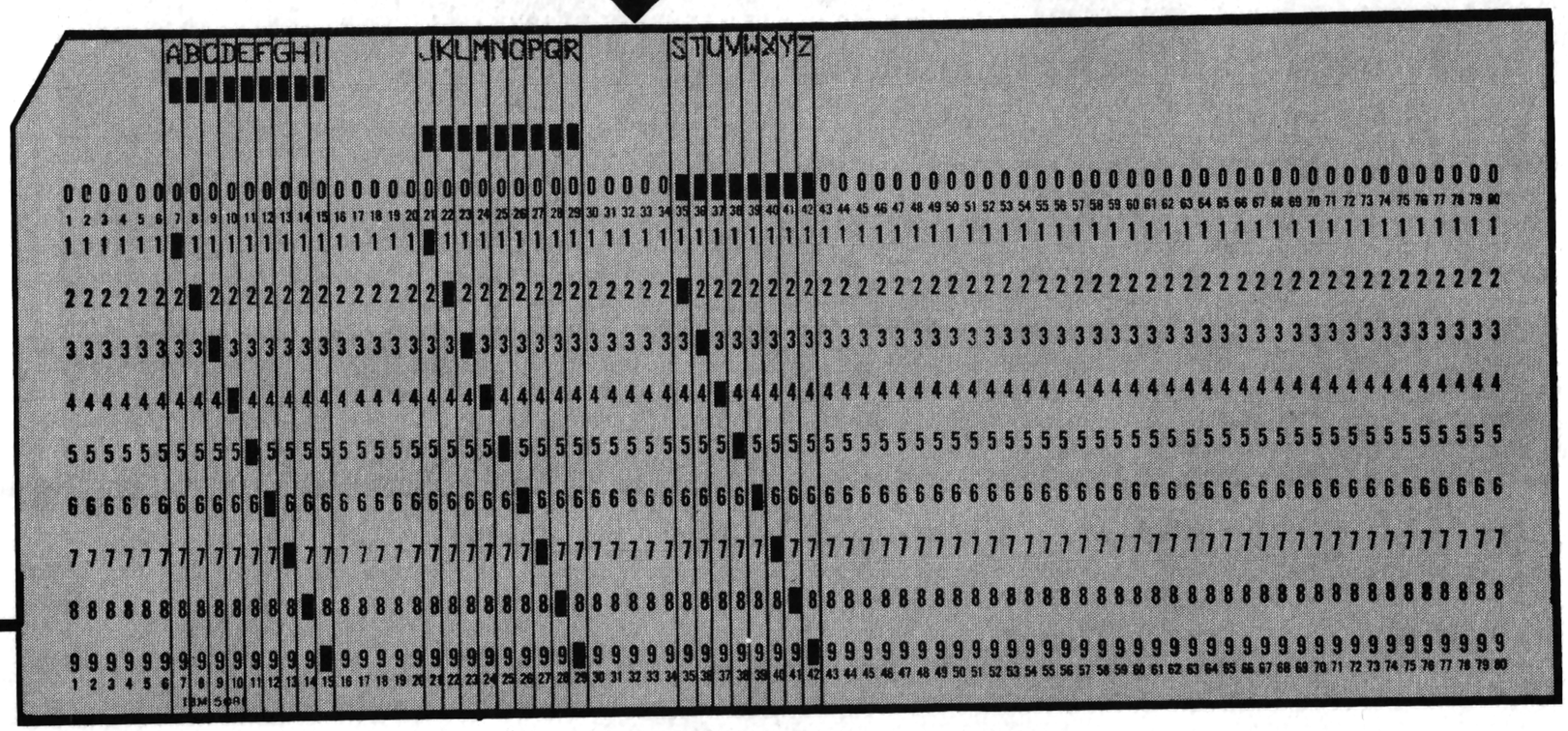
Numeric,

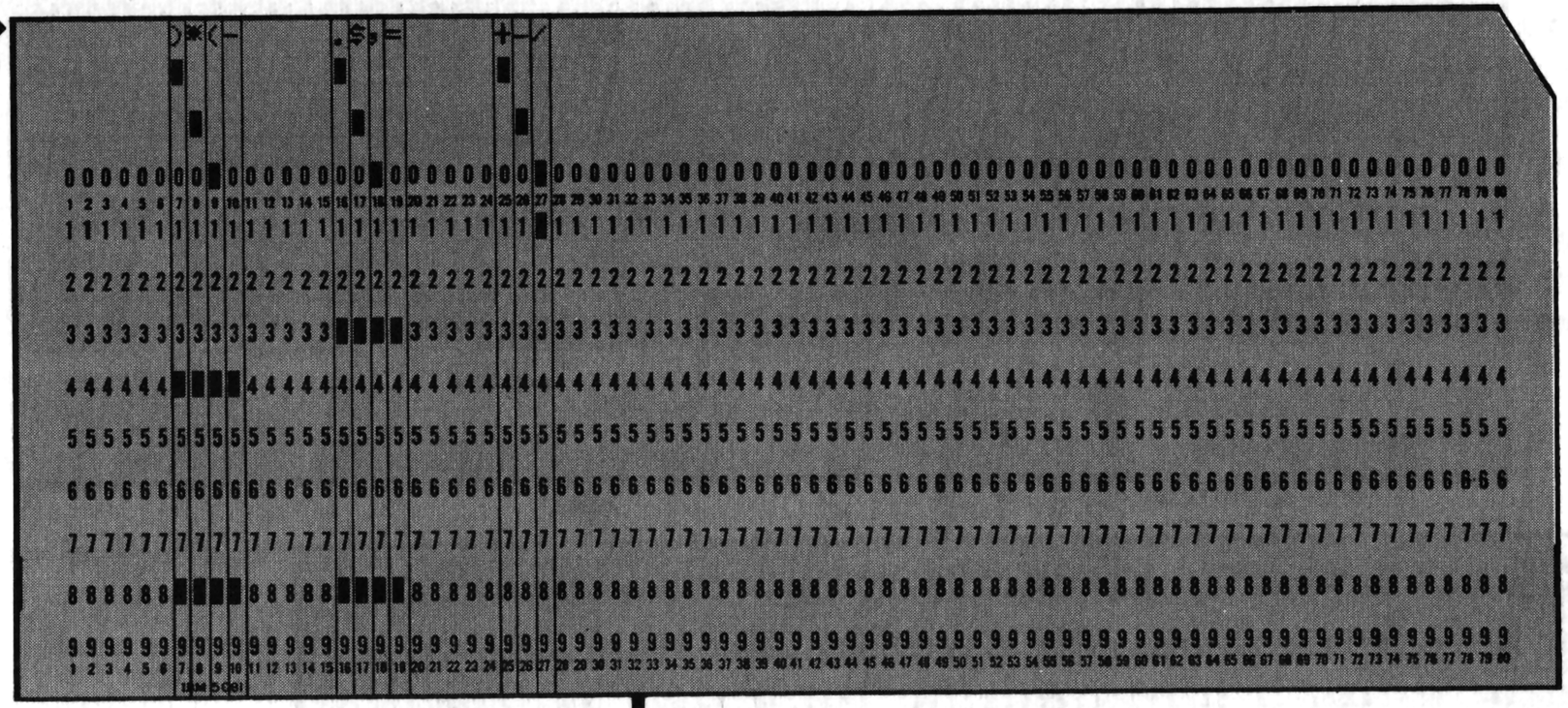
Alphabetic, and

Special Characters

punched in.







There are several alternative sets of special characters that may be used at different computer installations.

Exercises

1. Programs to be compiled are called _____ programs.

ANSWER ON P. 163

2. Source Programs must be compiled into _____ programs before they can be executed.

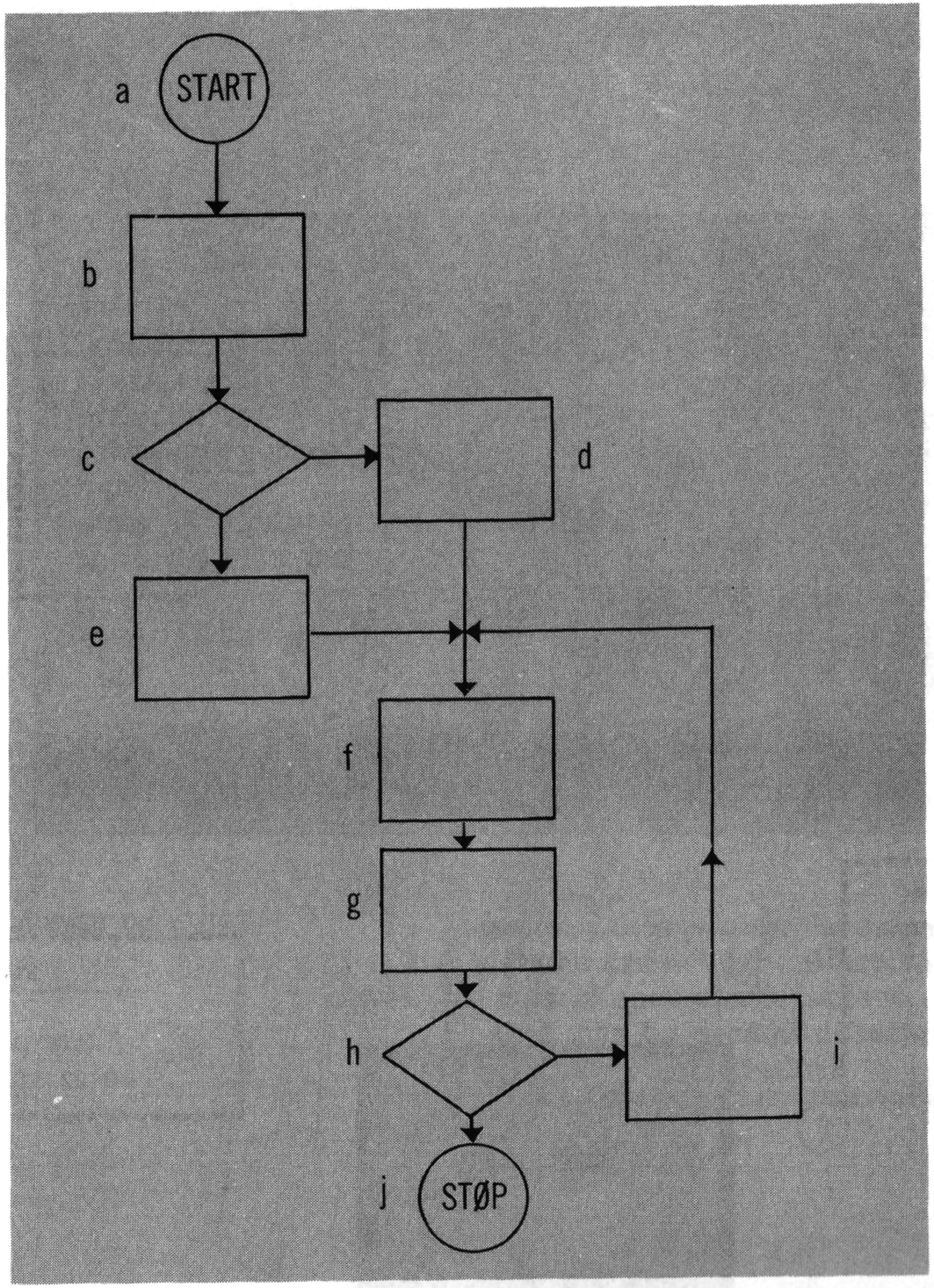
ANSWER ON P. 167

3. The process of producing a Fortran Object Program from a Source Program is called _____.

ANSWER ON P. 168

4. Executable statements are compiled into machine _____.

ANSWER ON P. 171



5

This program will not terminate until control reaches box _____.

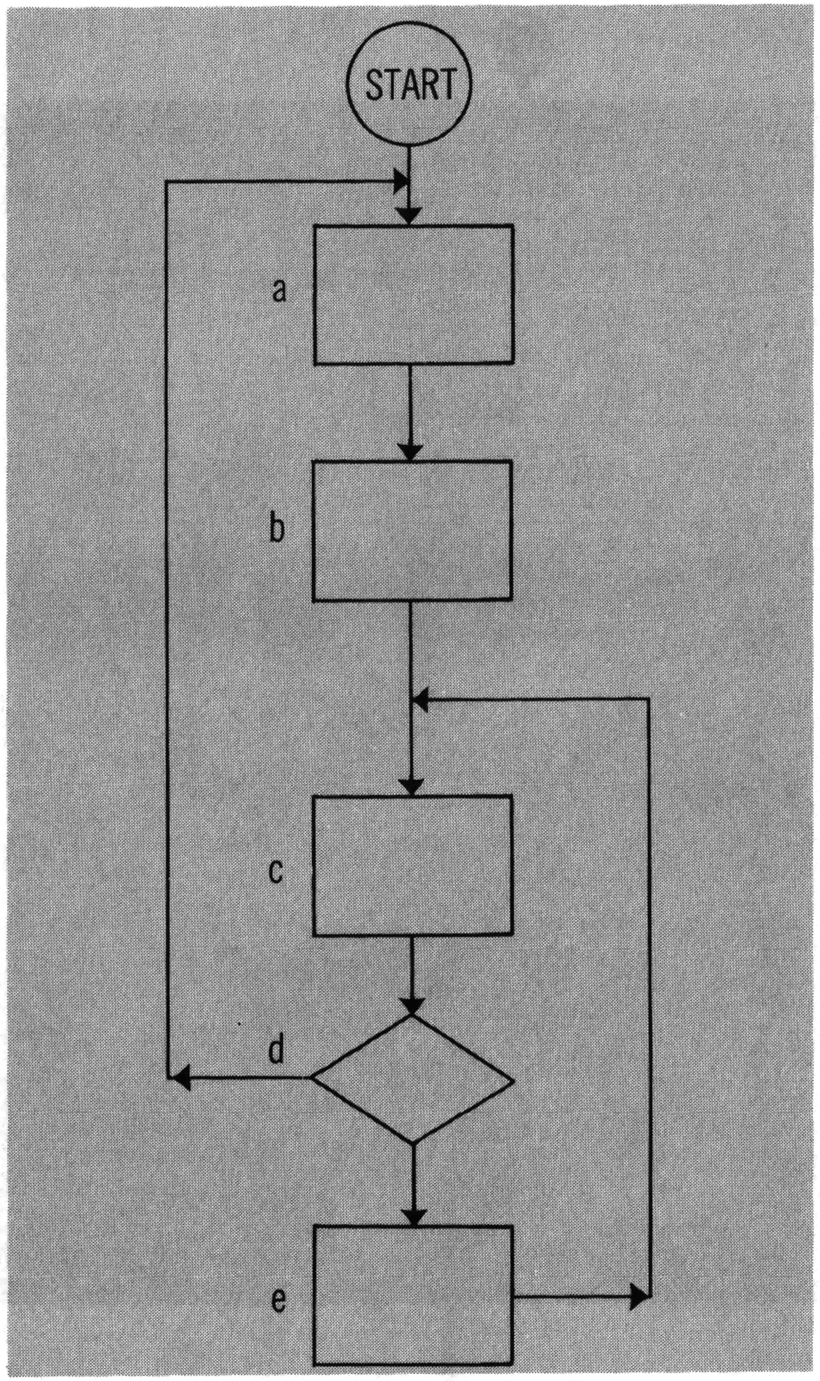
ANSWER ON P. 167

6

When will this program terminate?

1) Never
2) At box (e)
3) At box (d)

ANSWER ON P. 164



7

How often will control reach box (c)?

a) Never

b) Once

c) One-hundred times

ANSWER ON P. 172

8

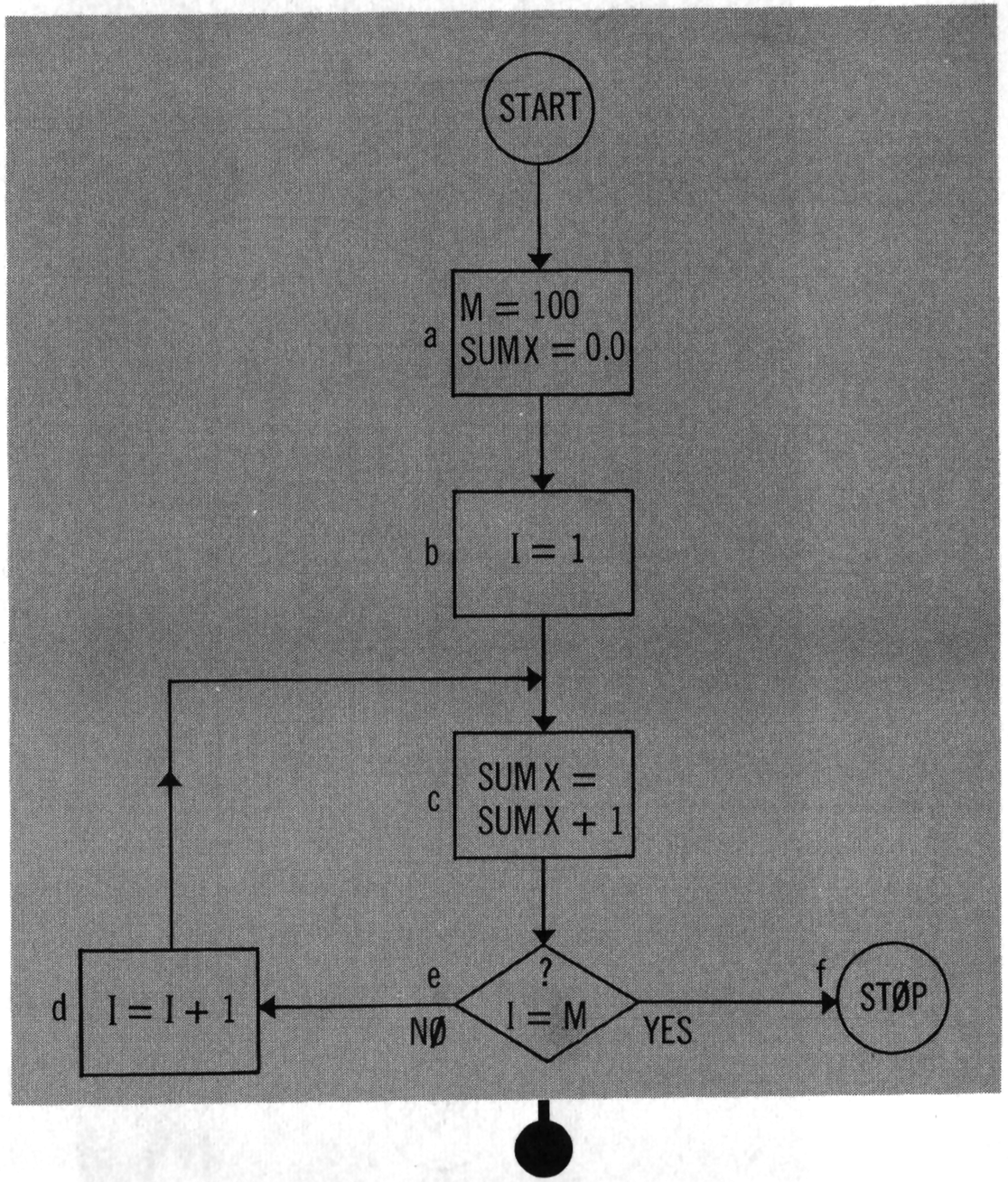
How many times will the contents of box (c) be executed?

a) Once

b) Never

c) "M" times

ANSWER ON P. 173



Part Two: PROGRAM STRUCTURE

A Job ● Subroutines ● Main Program ●
Interprogram Flow of Control ● Exercises ●
Main Program Task ● Diagnosis ●
Comment Statements

A Job

A "Job" consists of:

1. The Program Section
2. The Data Section
3. The System Control Cards

System Control Cards identify the Job, and regulate the preprocessing of the programs in it. The control cards required at computing installations are different from installation to installation.

The data are in one section and that section is usually marked by a particular System Control Card.

The Program Section contains all the related program segments that have been written to accomplish a particular task.

There must be one and only one Main Program in the program section. But there may be many subroutines—or none.

Most large, well-organized jobs consist of one Main Program and several subroutines.

Subroutines

A subroutine is a self-contained program that is compiled independently of other subroutines or the Main Program.

A set of Fortran statements is defined as a subroutine by the first statement in the set.

The general form of a subroutine-defining statement is

SUBRØUTINE Name

where "Name" may be any combination of from one to six alphabetic characters.

For example,

SUBRØUTINE DØIT

defines a set of statements as a subroutine whose name is

"DØIT".

The *end* of a subroutine is also marked with a special statement.

The last statement in a subroutine must be an "END" statement.

END

The compiler will compile all the statements between the SUBROUTINE statement and the END statement as a single subroutine.

The END statement says, "This is the last card in this program."

The termination of a subroutine must be the following statement:

RETURN

The RETURN statement will cause the flow of control (execution) to return to wherever it was before control was transferred to the subroutine.

A RETURN statement is the last *logical* statement.
The END statement is the last *physical* statement.

Thus, there are three crucial statements in every subroutine:

```
SUBRØUTINE Name
  . . .
  . . .   } other statements
  . . .
  . . .
RETURN
END
```

Main Program

Subroutines may be "called" from any point in a Main Program. The calling statement is

CALL Name

where "Name" is the name of the subroutine to which control is to be transferred.

A program is defined as a *Main Program* by the single fact that the first statement in it *is not* a subroutine-defining statement.

Thus, a program that is a Main Program may begin with any statement *except* a subroutine-defining statement.

The last statement in a Main Program must be an END statement:

END

The logical termination of a job must be a statement in the Main Program that calls a special System Subroutine which returns control to the Monitor System.

Although there are several Program-terminating System Subroutines, the one to use here is

CALL EXIT

subroutine EXIT terminates the job, and returns control to the Monitor System.

Interprogram Flow of Control

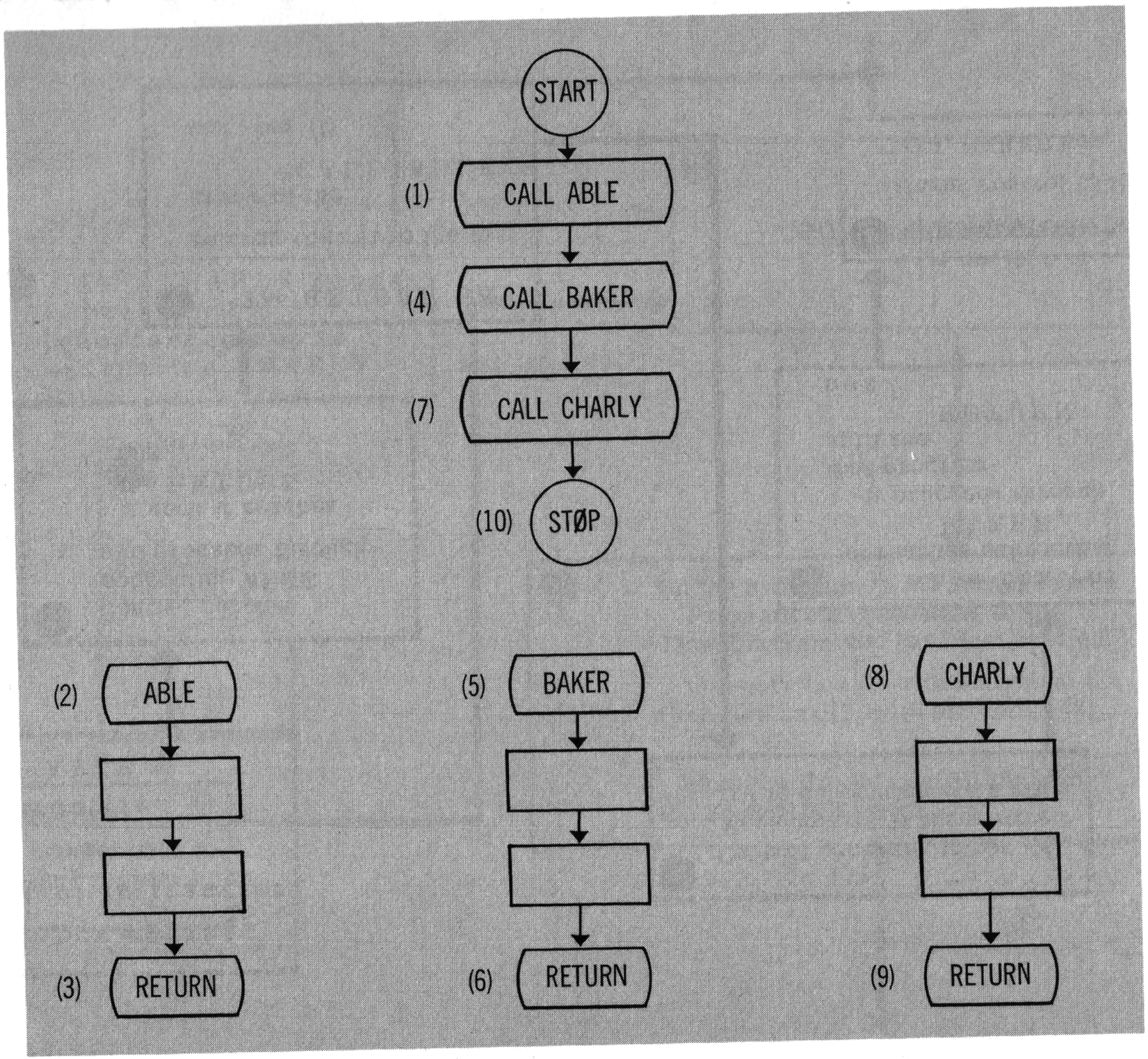
It is from the Main Program that subroutines are normally called.

Control flows from the Main Program to a subroutine, and then back to the Main Program.

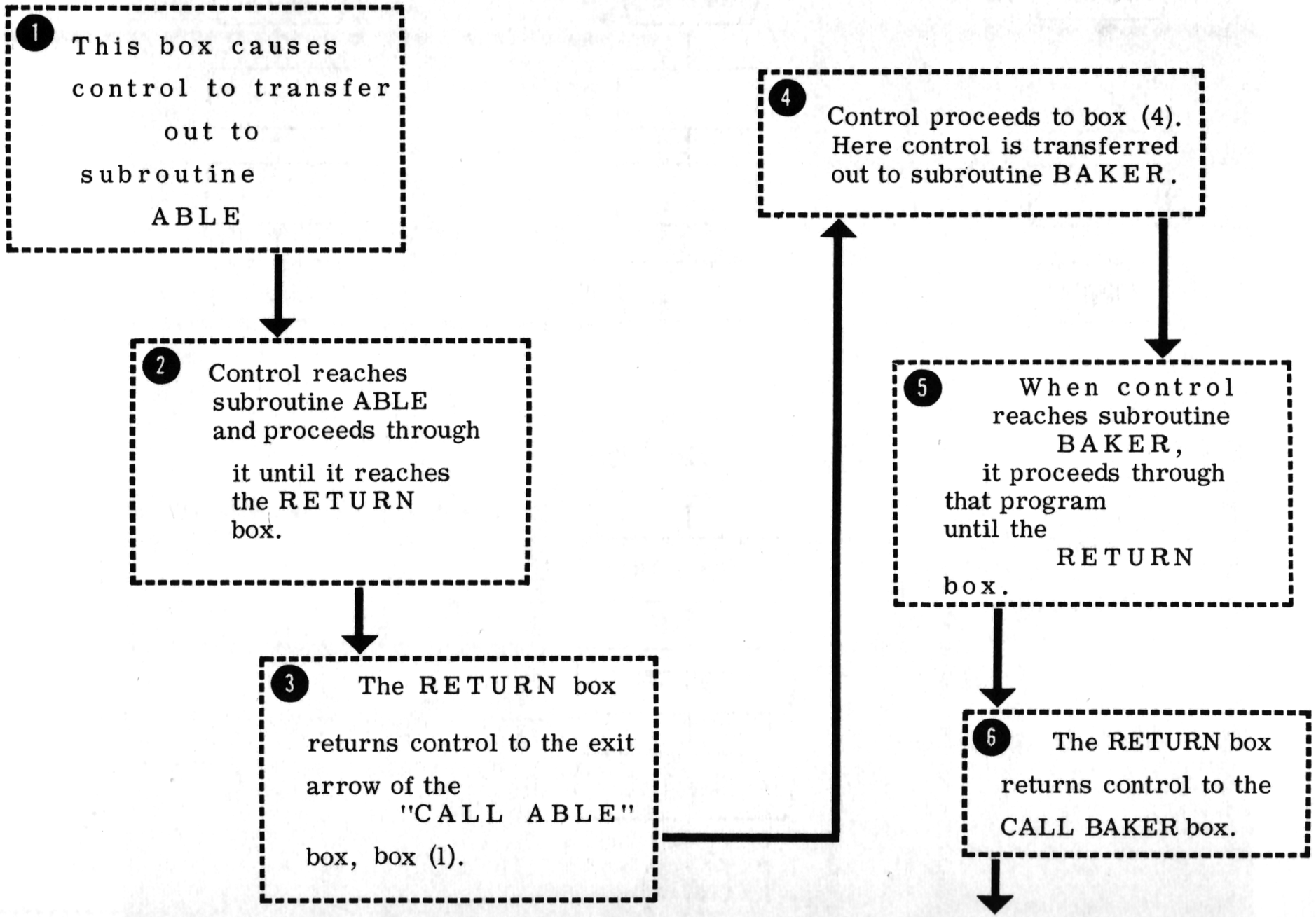
Subroutines may also be called from *other* subroutines.

Control could flow, for example, from a Main Program to a subroutine, and from there to a second subroutine.

The RETURN statement in the second subroutine returns control to the first.



Interprogram Flow of Control


1 This box causes control to transfer out to subroutine ABLE
2 Control reaches subroutine ABLE and proceeds through it until it reaches the RETURN box.
3 The RETURN box returns control to the exit arrow of the "CALL ABLE" box, box (1).
4 Control proceeds to box (4). Here control is transferred out to subroutine BAKER.
5 When control reaches subroutine BAKER, it proceeds through that program until the RETURN box.
6 The RETURN box returns control to the CALL BAKER box.

7 CALL CHARLY causes control to be transferred out to subroutine CHARLY.

8 Subroutine CHARLY operates.

9 The RETURN box sends control back to the Main Program.

10 "STØP" indicates the termination of the job.

11 On the next page is a Fortran Source Program corresponding to the Flow Diagram you have just studied.

Compare box (1) with statement (1). Compare box (2) with statement (2), etc., etc.

Note the flow of control through the Source Programs.

15

The correct statement in a Main Program to call a subroutine named BAKER is

```
RETURN BAKER
    GET BAKER
        CALL BAKER
```

ANSWER ON P. 162

16

Which of the following statements causes control to be returned to the Monitor System, thus terminating the program?

CALL EXIT END RETURN SUBRØUTINE

ANSWER ON P. 167

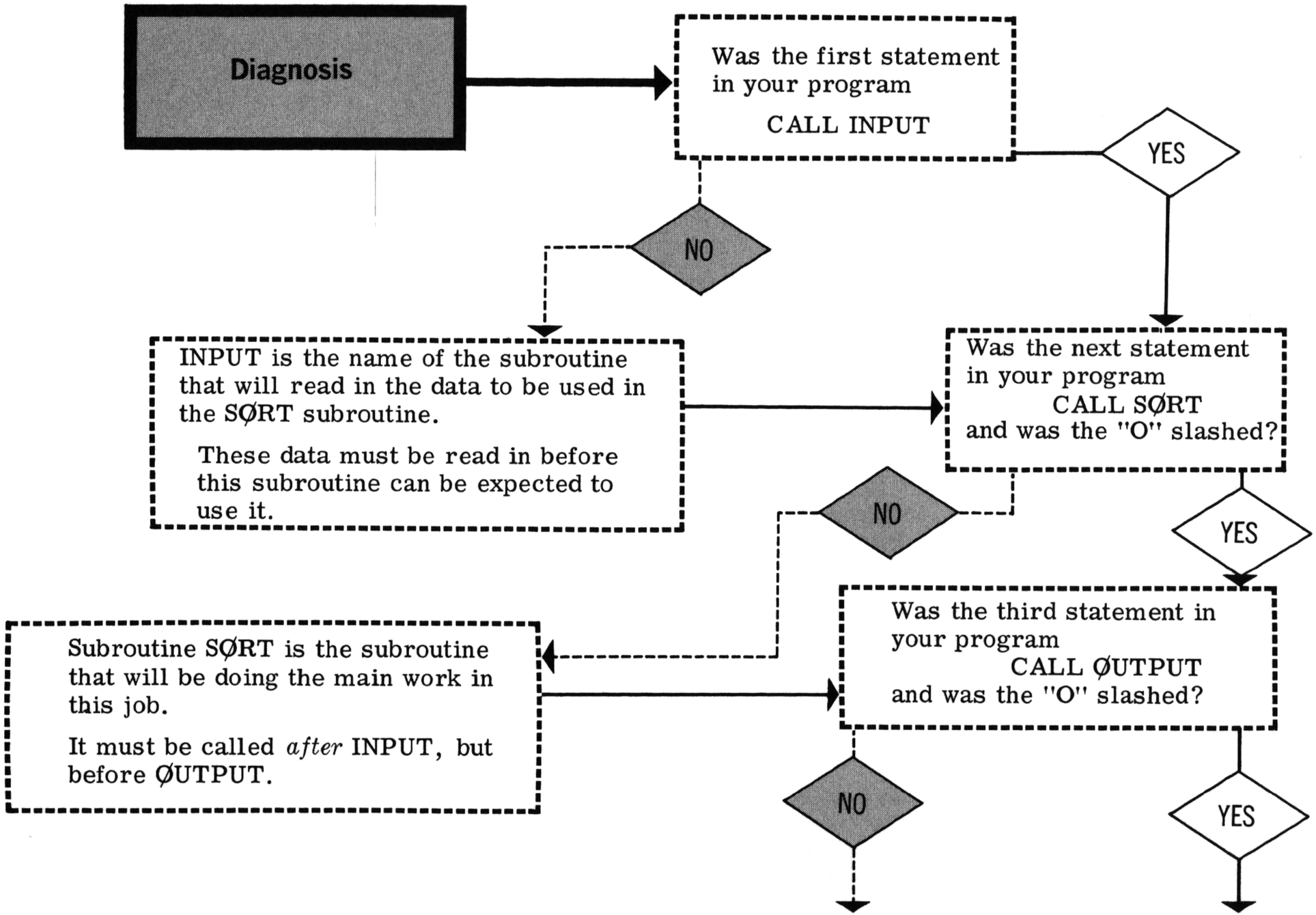
Main Program Task

Beginning on about the fifth line of a Fortran coding sheet, write a complete, but short, Main Program that calls each of the following subroutines in turn:

INPUT, SØRT, ØUTPUT, and EXIT.

SPECIAL INSTRUCTIONS: * * *

Do not continue until you have completed this task.


Diagnosis
Was the first statement in your program
CALL INPUT
YES
NO
INPUT is the name of the subroutine that will read in the data to be used in the SØRT subroutine.
These data must be read in before this subroutine can be expected to use it.
Was the next statement in your program
CALL SØRT
and was the "O" slashed?
NO
YES
Subroutine SØRT is the subroutine that will be doing the main work in this job.
It must be called *after* INPUT, but before ØUTPUT.
Was the third statement in your program
CALL ØUTPUT
and was the "O" slashed?
NO
YES

Part Three: VARIABLES AND CONSTANTS

Fortran Variables ● Exercises ● Fortran Constants ● Exercises ● Subscripted Variables ● DIMENSION Statements ● Exercises ● Common Storage ● Exercises

Fortran Variables

The heart of every Fortran program is the arithmetic calculations that are performed.

These calculations are done with numbers.

Numbers may be written in Source Programs as

Fortran Constants or they may be input data.

Every numerical value that is input data must be "assigned" to a Fortran variable.

Each Fortran variable has a specific storage location in memory for its numerical value.

Every numerical value that is read in by a program is stored in the memory location corresponding to the variable to which it is assigned.

Thus, a Fortran variable is like a symbolic "tag" for a storage location.

The value of the variable at any moment during execution is the value of the number in its storage location.

Before Fortran variables can be used in a calculation, they must be "defined."

A variable is said to be defined if it occurs in a Fortran statement that stores a number in the location named by the variable.

This rule ensures that a variable will have a value *before* it is used.

Variables may be defined

in Input Statements,

in DØ-loop Statements,

and as the name of the location where the result of an arithmetic calculation should be stored.

Whenever a variable is used in a program, it is the storage location that it names that will be affected.

The actual storage location named by a particular variable cannot be known until after the program is compiled.

And while the program is being written, it is not necessary to know the actual storage location.

If the first character is
A, B, C, D, E, F, G, H, --- Ø, P, Q,
R, S, T, U, V, W, X, Y, Z,
the variable is FLOATING POINT.

For example, the following variables are

Fixed Point Mode

I J K L M N

MAN KØUNT ITEMNØ NUMBER

JOBNØ NANCY

LUCK MANNØ LUCY JANE

On the other hand, these
variables are all

Floating Point Mode

X P Q A F R DØG Y

TØM TABLE TABØNE TABTWØ

GIZMØ

PETER ZENITH RATE RCA ETC

Exercises

(17) Invent the names of five Fixed Point variables.

(Write them on scratch paper.)

ANSWER ON P. 164

(18) Invent the names of five *Floating Point* variables.

(Write them on scratch paper.)

ANSWER ON P. 167

Fortran Constants

To be able to use a number in a program, it is not always necessary to invent a Fortran variable for it, and then read in a value.

Numbers may be written as decimal numerals in Fortran statements.

Subscripted Variables

The Fortran variables you have learned about thus far have been single entities.

They are called "nonsubscripted" variables to distinguish them from another type of variable, the "subscripted" variable.

Nonsubscripted variables are automatically assigned *one* storage location by the compiler, when it compiles the Source Program into an Object Program.

Subscripted Variables, on the other hand, are assigned as many storage locations as you specify in the program.

NOTE: A variable with one subscript (a single subscripted variable) is called a One-dimensional Array.

A variable with more than one subscript is possible, but will not be discussed here.

DIMENSION Statements

The statement that must be used to assign storage locations to a subscripted variable is called a DIMENSION statement.

This statement instructs the Fortran Compiler to reserve a specified number of locations for each subscripted variable in the program it is compiling.

The general form of a DIMENSION statement is

DIMENSION $v_1(n_1)$, $v_2(n_2)$, . . .

"v" is any legal Fortran variable

"n" is a Fixed Point *Constant* specifying the number of storage locations to be reserved or allocated to "v."

Note:
Several variables may be "dimensioned" in one DIMENSION statement.

Because the storage allocation for any subprogram is built into the Object Program,

any change in the storage requirements for that subprogram can be made only by

changing the DIMENSION statement in the Source Program, and then recompiling that Source Program.

For this reason, Subscripted Variables should be dimensioned for the maximum number that will ever be needed for that variable.

Common Storage

You have already learned that

1. Variables are allocated storage locations by the Fortran Compiler,
2. Nonsubscripted Variables are allocated one storage location, and
3. Subscripted Variables are allocated as many locations as you specify in the DIMENSION statement.

Actually the compiler allocates storage locations in either of two ways:

1. as a relative address, internal to the program in which the variable is used, or
2. as an absolute address in a special part of memory called Common.

This technique allows the compiler to compile

one subprogram completely independent from any other subprogram.

The compiler does not need to keep track of the

relative addresses of internal

variables from one subprogram to the next,

even within the same job.

And the external, absolute addresses assigned to Common Variables are always assigned in a systematic way starting at the same absolute location.

Thus, the problem of communicating values from one subprogram to the next is not a problem solved by the compiler. It is your responsibility, and must be a part of the design of your program.

You may communicate the values of internal variables from one subprogram to another by means of arguments in a calling sequence; a complicated technique reserved for a future learning experience.

You may communicate Common Variables by listing them in COMMON statements in the subprograms between which you want them communicated.

If a variable is not listed in a COMMON statement, it is an Internal Variable.

Variables that are listed in a COMMON statement are called Common Variables.

The general form of a COMMON statement is

COMMON v_1, v_2, v_3, . . .

The "v's" are variables.

31

If the nonsubscripted variables, A, B, and C are used in Subroutine PETER, write the COMMON statement that will assign them to the first three Common Storage locations.

ANSWER ON P. 168

32

If A, B, and C are also to be used in Subroutine HELP, *write* the COMMON statement for HELP that will also assign them to the first three locations in Common.

ANSWER ON P. 171

33

If A, B, and C are also to be used in the Main Program that calls PETER and HELP, write the COMMON statement for A, B, and C.

ANSWER ON P. 173

34

If Subroutine SPECAL uses only C write a COMMON statement for SPECAL.

ANSWER ON P. 164

35

How many Common Storage locations will be allocated by the following pair of statements?

```
CØMMØN X, Y, Z
DIMENSIØN X(100), Y(100), Z(100)
```

ANSWER ON P. 168

36

How many Common Storage locations will be allocated by these two statements?

```
CØMMØN AJAX, BILLY, LAZY
DIMENSIØN BILLY (100)
```

ANSWER ON P. 165

Part Four: INPUT STATEMENTS

Input Data Fields ● Input FORMAT Description ● Multiple Fields ● READ Statements ● Relations between Fields, Descriptions, and the Variable List ● Exercises ● Input Task and Diagnosis

Input Data Fields

The transmission of numerical values into and out of the computer during the execution of a program is controlled by Input and Output instructions.

In their original form, these numerical values are punched on tabulating cards. Each digit of the numerical value is punched in one column.

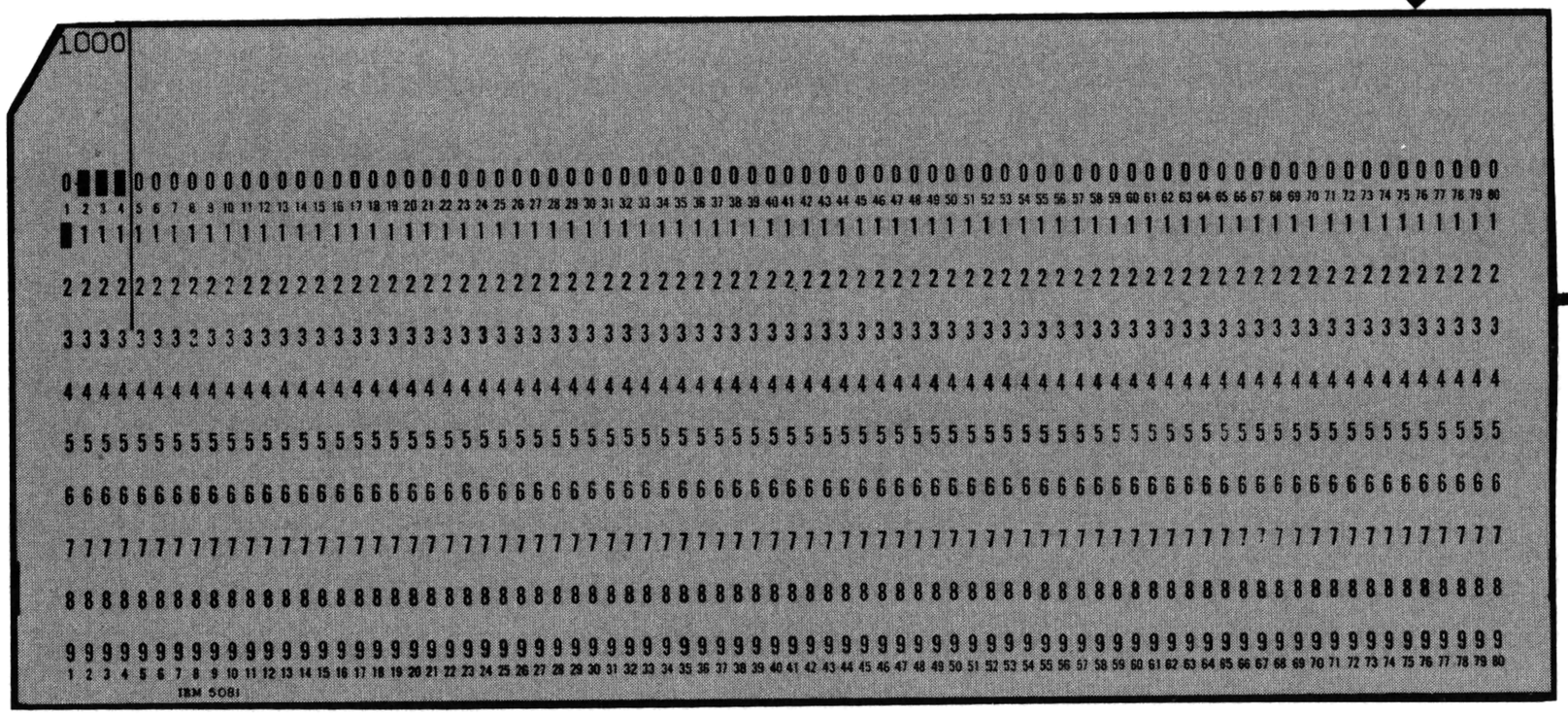
There may be only one, or there may be several numerical values, punched on one tabulating card.

For purposes of description, each value is said to occupy one "data field."

A tabulating card containing several numerical values is described as having several data fields.

Numbers may also be read in that are to be value (s) of *Fixed Point* variables.

For example, if 1000 is to be read in as the value of "k," it could be punched in the first four columns of a card.



The description of this data field would be

I4

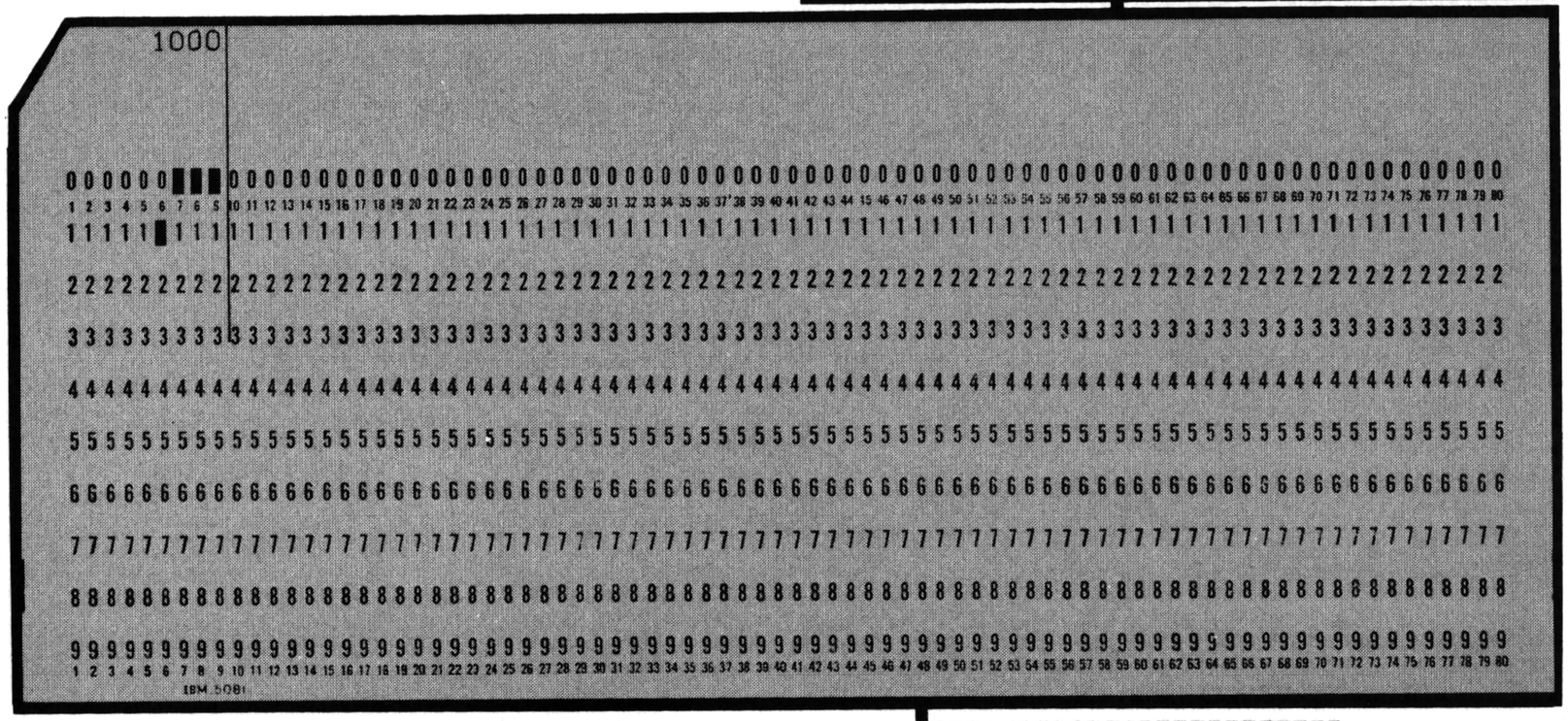
where "I" specifies *Fixed Point* (Integer) mode; and where "4" specifies four columns in the data field.

In a FORMAT statement it would be written as

FØRMAT (I4)

There may be more columns in a data field than are needed to contain the number in that field.

If the number is Fixed Point, the field should be described so that the number will be "right-justified" within it. The extra columns should be at the *left* of the number.



should be read in with an I9 data field description. The unused columns are treated as zeros. Zeros to the left of a Fixed Point number will *not* effect its value. Zeros to the right will increase its value by factors of 10.

Several data fields in a row, having the same description, may be described by multiplying the description.

The three data field descriptions,

F10.3, F10.3, F10.3

can be written as

3F10.3

The "3" before the "F" means that there are three fields described.

The data field descriptions for a card should be written in a FORMAT statement in the Fortran program that reads the card:

FØRMAT (I4, 3F10.3)

This statement is *not executed*. Its only purpose is to supply information for the statement which, when executed, actually causes the card to be read.

READ Statements

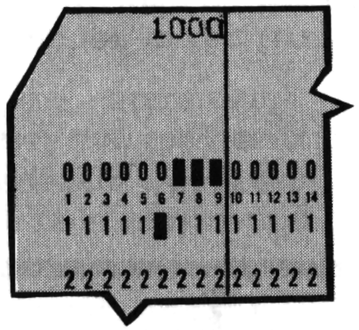
The statement which, when executed, causes the card to be read is a READ statement.

The general form of a READ statement is

READ n, Variable List

where "n" is the number of the FORMAT statement that *describes* the *card* to be read, and where the "Variable List" is a list of the *variables* where the numbers are to be *stored*.

Thus, to read this card,

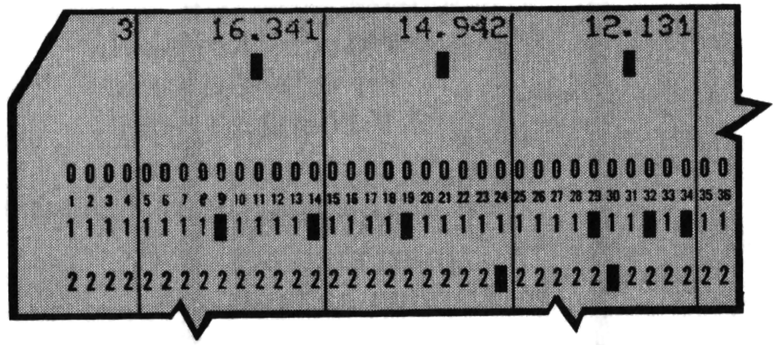


the following FØRMAT and READ statements could be used:

```
3   FØRMAT (I9)
    READ  3, N
```

The effect of the execution of this READ statement will be to store *one thousand* for the value of N.

This card could be read in with these statements:




```
4   FØRMAT (I4, 3F10.3)
    READ  4, N, X, Y, Z
```

After execution of the READ statement
N = 3 X = 16.341
Y = 14.942 and Z = 12.131

Relations between Fields, Descriptions, and the Variable List

There is a one-to-one positional correspondence between the variables on the list and the description in the FORMAT statement.

The number in the first data field, the one described by the first description, is stored in the first variable on the variable list.

The numerical value in the *second data field* becomes the value of the *second variable* on the list.

The third value is assigned to the third variable, etc.

This process continues until *every* variable on the list has a value assigned to it.

The READ statement will cause a value to be read in for every variable on the variable list.

Each variable requires a FORMAT description.

There should be agreement of mode.

Integer number in the data field; Fixed Point (I) description in the FORMAT statement; Fixed Point variable in the list.

Floating Point number in the data field; Floating Point (F) description in the FORMAT statement; Floating Point variable in the list.

Several cards can be read with the same FORMAT statement during the execution of one READ statement.

The READ statement continues to read in numbers as long as there are variables on its *list* for which no values have yet been read in.

Descriptions in the FORMAT statement apply consecutively to variables on the list.

If the last description has been applied and if there are still more variables on the list, the FORMAT descriptions are repeated from the left parenthesis, but apply to the next card following the one just read.

Values for subscripted variables may be read in the same way.

For example,

```
33  FØRMAT (F10.3, F8.2)
    READ 33, A, B, C, D, E, F, G, H
```

would read in a value for "A" and "B" from the first card, then return to the beginning of the FORMAT statement and read in two values from the second card for "C" and "D", two for "E" and "F" from the third card, and for "G" and "H" from the fourth card.

For example,
to read in values for the subscripted variable X(I), where the subscript "I" runs from one to fifteen, the following READ statement could be used:

```
11  . . .
    READ 11, X(1), X(2), X(3), X(4), X(5),
    X(6), X(7), X(8), X(9), X(10), X(11),
    X(12), X(13), X(14), X(15)
```

But this rather lengthy (and tiring) procedure can be shortened considerably by using a technique called "List Control."

When using subscripted variables, the number of values that will be read in is determined by the *range* of the subscript. This range is specified in the List Control.

List Control is, therefore, a special technique that can be used to read in the values of a subscripted variable.

The general form of List Control is

(x(i), i = m, n)

where x(i) is the subscripted variable and

i = m, n is the range of the subscript

In a READ statement, values will be read in and stored in "m" and succeeding locations until and including "n."

For example,

```
44      FØRMAT (7F10.3)
        READ 44, (X(I), I = 1, 7)
```

will read in eight values from the data card for X(1), X(2), X(3), X(4). . . X(7), respectively.

Using this combination,

```
66 FØRMAT (7F10.3)
   READ 66, (X(I), I = 1,700)
```

seven values will be read from the first card for X(1) through X(7).
Repeating the FORMAT description and reading from the next card, seven more values will be read for X(8) to X(14), etc.

This process of reading from the next card and repeating the FORMAT description will continue until the List is exhausted. The READ statement will continue to be executed until X(700) has a value read in for it.

At that point the READ statement will be finished, and control will pass to the next statement.

Exercises

37

Whenever a READ statement is executed, it reads values from the next data card according to the descriptions specified in a _____ statement.

ANSWER ON P. 171

38

On a tabulating card, a group of columns containing numbers to be read in as one numerical value is called a _____ field.

ANSWER ON P. 173

39

One data field is read in for every _____ on the Variable List.

ANSWER ON P. 173

40

How many *data fields* will be read by a READ statement that has *five* variables on its Variable List?

ANSWER ON P. 175

41

How many *data fields* will be read by this READ statement?
READ 62, A, B, C

ANSWER ON P. 172

42

FORMAT statements describe the _____ _____ on data cards.

ANSWER ON P. 171

43

How many data fields are on the card described by this statement?

62 FØRMAT (3F10.4)

ANSWER ON P. 165

(44)

How many *data cards* will be read by these statements?

```
62  FØRMAT (3F10.4)
    READ 62,  A, B, C
```

ANSWER ON P. 172

(45)

How many data *fields* will be read by a READ statement that has one thousand variables on its Variable List?

ANSWER ON P. 173

(46)

How many data *fields* will be read by this statement?

READ 23, (Z(K), K = 1,100)

ANSWER ON P. 172

(47)

How many data fields will be read in by this statement?

READ 65, (A(M), M = 1,N)

ANSWER ON P. 175

(48)

How many data *fields* are on the card described by this statement?

```
23  FØRMAT (10F 8.3)
```

ANSWER ON P. 169

(49)

How many data *cards* will be read by these statements?

```
23  FØRMAT (10F8.3)
    READ 23, (Z(K), K = 1,100)
```

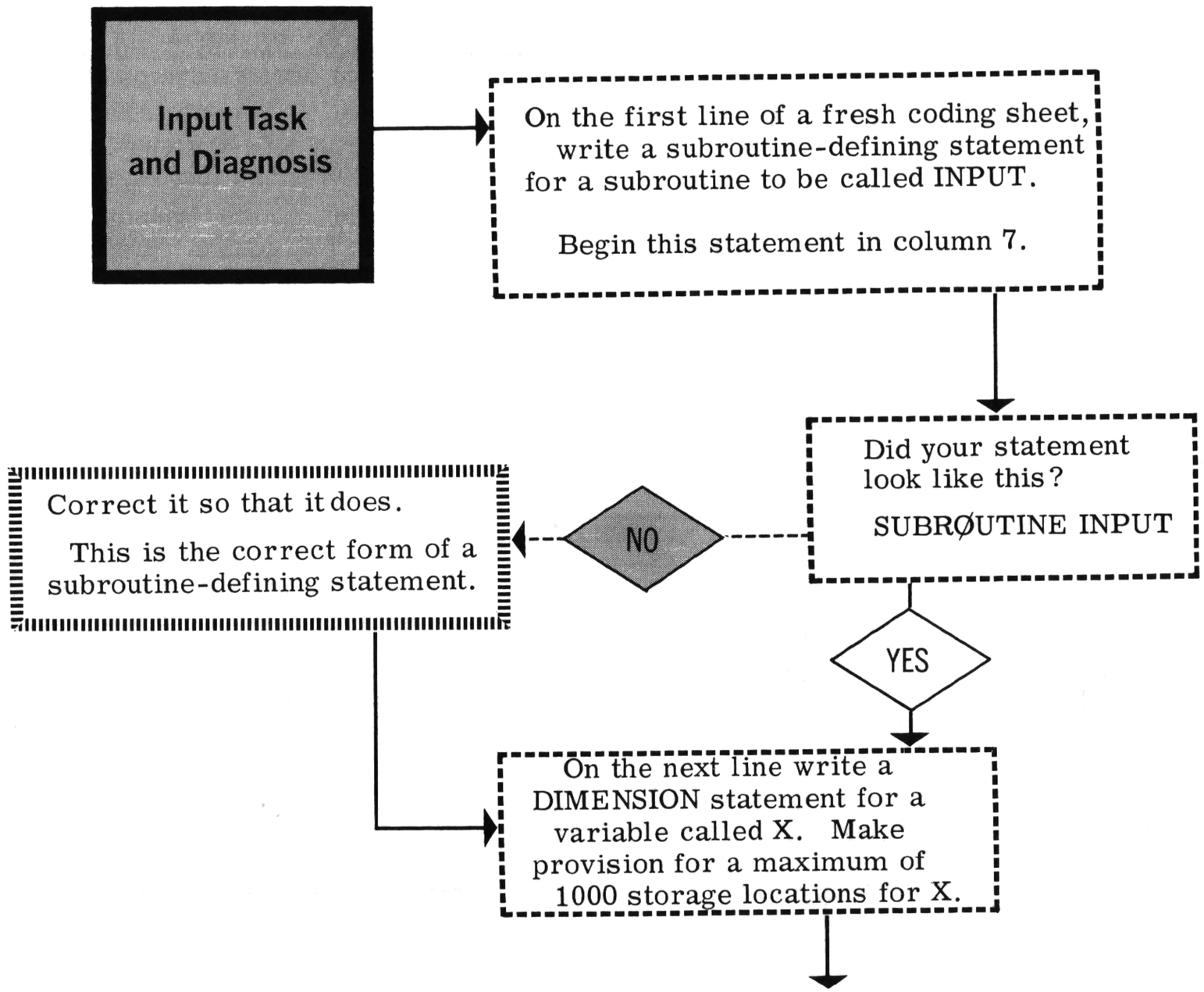
ANSWER ON P. 171

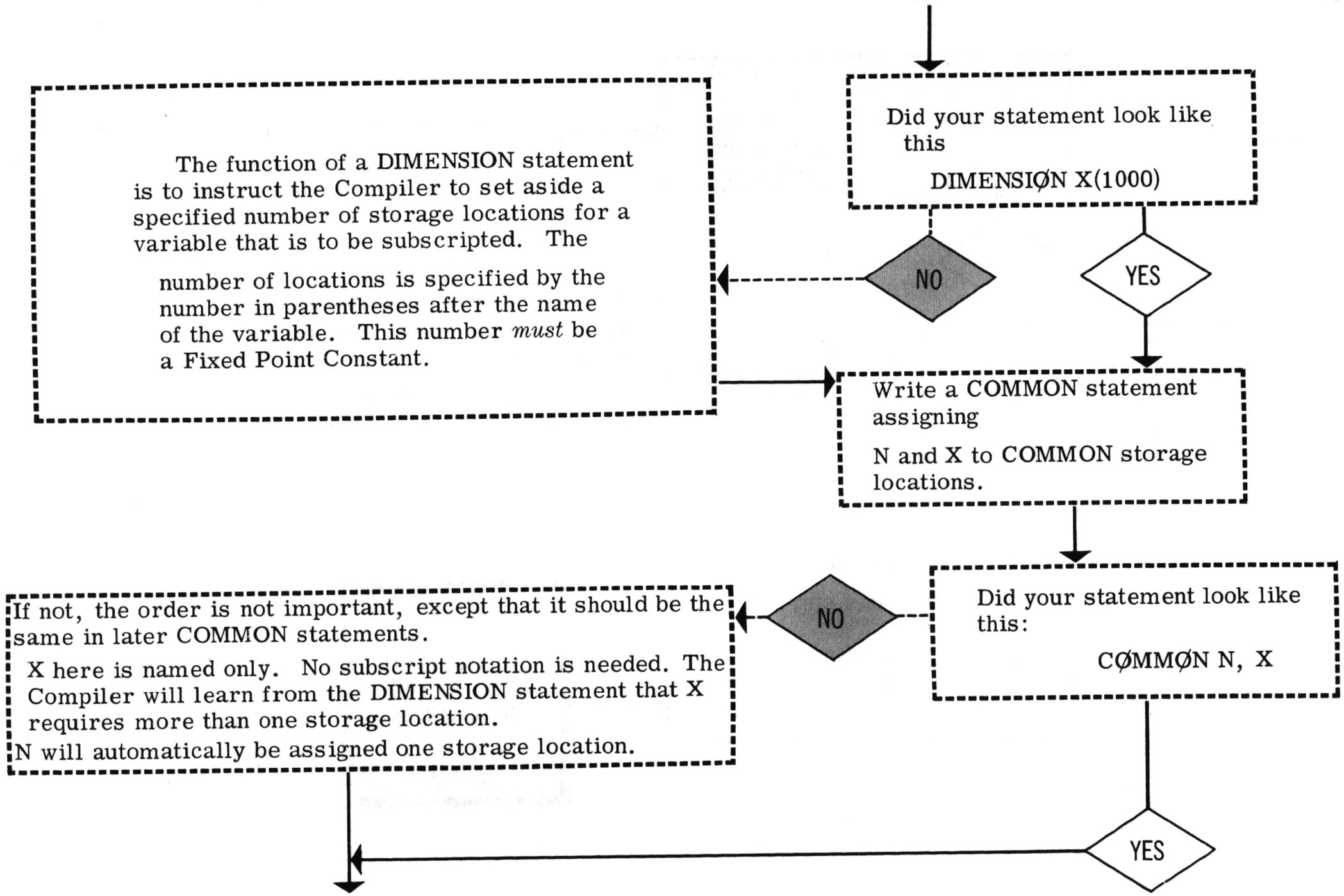
(50)

How many data *cards* will be read by these two statements?

```
44  FØRMAT (I4)
    READ 44, I, J, K, L, M, N
```

ANSWER ON P. 175


Input Task and Diagnosis
On the first line of a fresh coding sheet, write a subroutine-defining statement for a subroutine to be called INPUT.
Begin this statement in column 7.
Did your statement look like this?
SUBRØUTINE INPUT
NO
Correct it so that it does.
This is the correct form of a subroutine-defining statement.
YES
On the next line write a DIMENSION statement for a variable called X. Make provision for a maximum of 1000 storage locations for X.


Did your statement look like this
DIMENSIØN X(1000)
NO
YES
The function of a DIMENSION statement is to instruct the Compiler to set aside a specified number of storage locations for a variable that is to be subscripted. The number of locations is specified by the number in parentheses after the name of the variable. This number *must* be a Fixed Point Constant.
Write a COMMON statement assigning
N and X to COMMON storage locations.
Did your statement look like this:
CØMMØN N, X
NO
If not, the order is not important, except that it should be the same in later COMMON statements.
X here is named only. No subscript notation is needed. The Compiler will learn from the DIMENSION statement that X requires more than one storage location.
N will automatically be assigned one storage location.
YES

Write and number a FORMAT statement that describes a card with one Fixed Point data field having a width of four columns.

Next, write a READ statement that reads in the value of a variable called N, according to the FORMAT statement you just wrote.

Do your last two statements look something like this:

```
63  FØRMAT (I4)
    READ 63, N
```

NO

YES

The READ statement must refer to a FORMAT statement by using the number of that FORMAT statement. In the "correct" answer the FORMAT statement was number 63. Your answer may have a different number. This is all right. If necessary, repair your statements.

Assume that the values to be read in for X are already punched in tabulating cards.

Assume that there are three digits in the fraction, that each data field is eight columns wide, and that there are ten data fields on a card. Now write and number a FORMAT statement to be used to read these values.

Now write a READ statement, referring to the FORMAT statement you just wrote. Using List Control, specify that N values are to be read in for the subscripted variable X(I).

Do your FORMAT and READ statements look something like this:

```
15  FØRMAT (10F8.3)
    READ 15, (X(I), I = 1, N)
```

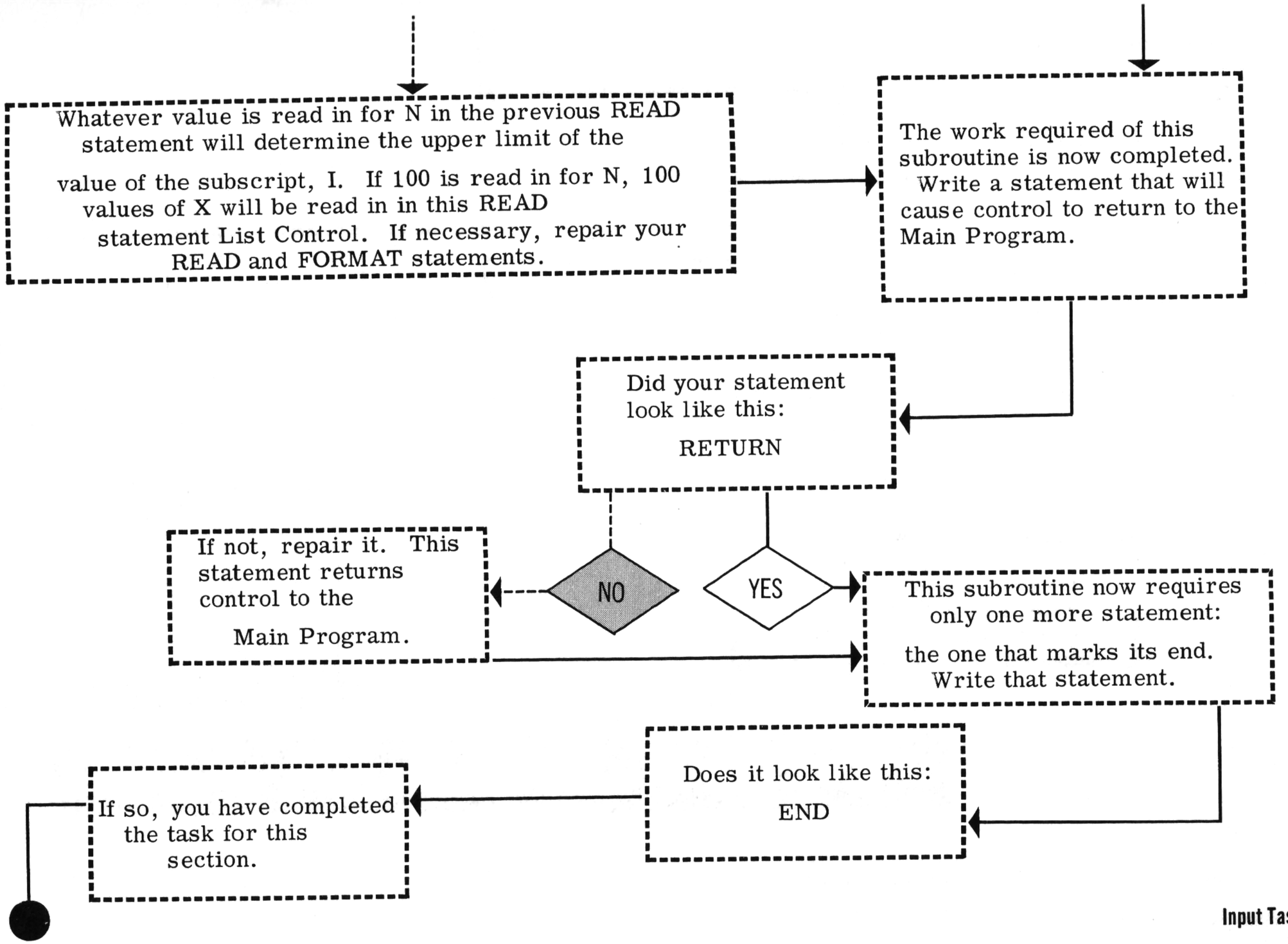
NO

YES

The FORMAT statement number must be the same as the reference to it in the READ statement.

The number of values to be read in by the READ statement is specified by List Control.

(X(I), I = 1, N)


Whatever value is read in for N in the previous READ statement will determine the upper limit of the value of the subscript, I. If 100 is read in for N, 100 values of X will be read in in this READ statement List Control. If necessary, repair your READ and FORMAT statements.
The work required of this subroutine is now completed. Write a statement that will cause control to return to the Main Program.
Did your statement look like this:
RETURN
NO
YES
If not, repair it. This statement returns control to the Main Program.
This subroutine now requires only one more statement:
the one that marks its end. Write that statement.
Does it look like this:
END
If so, you have completed the task for this section.

Part Five: ARITHMETIC EXPRESSIONS

Expressions ● The Hierarchy of Operations ● Parentheses ● Exercises ● Mixed Expressions ● Exercises

Expressions

An *expression* is a series of variables and/or constants connected by arithmetic operation symbols.
A + B - ALPHA (9) ** 2 is an arithmetic expression.

A numerical value of an expression is calculated during the execution of the Object Program.
The result of this calculation is *one* number.

The following arithmetic operations may be specified in an expression:

FORTRAN SYMBOL	OPERATION
**	Exponentiation
/	Division
*	Multiplication
-	Subtraction
+	Addition

The following examples are expressions:

P ** V	PETER ** JØHN
T / D	3.1416/PEI
E * F	89.65 ** ALPHA
A - B	B - 3.975
X + Y	S + 2.0

The Hierarchy of Operations

In an expression, there may be any number (within reason) of variables and/or constants connected by operation symbols:

A + B * C / D ** F + G ** 2.876 - PETER

$$\left(A + B \times \frac{C}{D^F} + G^{2.876} - \text{PETER}\right)$$

in normal algebraic notation.

Or, there may be just one variable, BAKER, or one constant, 89.64.

In calculating the value of an expression, only one arithmetic operation is done at a time. The order in which they are done is (and must be) specified in the expression itself.

The rules governing the order of operations in expressions are called:
The Hierarchy of Operations.

The Hierarchy of Operations rules state that in every arithmetic expression:

Exponentiation (**) is done *first*,

Multiplication (*) and Division (/) *second*,

and

Addition (+) and Subtraction (-) *last*.

This means that in the expression,

A + B/C

the division of B by C is done first. The result of that division is then added to A.

In the expression,

X - Z * A ** F

the exponentiation, A **F, is calculated first, the result is multiplied by Z, and the result of that multiplication is subtracted from X.

There are three "levels" in the Hierarchy of Operations:

(1) Exponentiation

(2) Multiplication and Division

(3) Addition and Subtraction

Each first level operation (exponentiation) in an expression is done first, *from left to right*.

Operations at the second level (Multiplication and Division) are done next, *from left to right*.

Operations at the third level (Addition and Subtraction) are done next, from *left to right*.

In the expression A * B/C * D/E all the operations are second level; therefore, the first multiplication (A * B) is done first. This result is divided by C. This result is multiplied by D. This result is divided by E.

Thus, it is possible (and advisable) to analyze precisely the order in which each level of operation will be done in any expression.

Parentheses

To alter the order indicated by the Hierarchy of Operations, *parentheses* may be added. Those parts of an expression that are enclosed in parentheses will be calculated before those that are not.

Thus, in the expression,

(A + B)/C

A will be added to B first even though it is a third level operation, because it is enclosed in parentheses. This sum will be divided by C second.

In the expression

A ** (X - Z)

the Z will be subtracted from X first. This difference will then be the power to which A is raised.

There may be more than one operation inside a pair of parentheses:

(A + B/C)** D

In this case, the operations inside the parentheses will be done in the order indicated by the Hierarchy of Operations: B/C first; added to A second, raised to D third.

The Hierarchy of Operations inside a pair of parentheses is the same as the Hierarchy of Operations outside the parentheses. And, the Hierarchy inside may be altered by using an additional set of paired parentheses:

((A + B)/C) ** D

Operations within the *innermost* pair of parentheses are performed first, as indicated by the Hierarchy of Operations.

The next innermost next, etc., until the expression is one numerical value.

The way most programmers use the parentheses rule and the Hierarchy of Operations is this:

first, they write an expression that they believe will accomplish the operations they desire;
then they analyze it to determine whether it really does.

Finally, they add parentheses or otherwise change the expression so that it will do whatever is originally intended.

In writing an expression be careful not to have two operations in a row.

For example,

A + - B *is illegal*.

To repair it, parentheses are added:

A + (-B) is *legal*.

The operation symbols in a Fortran expression are *operations*, and never the sign of a variable or constant.

Therefore, (-B) should *not* be thought of as a negative number. The *minus* symbol in this example is an *operation symbol, not a sign.*

Thus,

(-B) is compiled as though there were a zero before the minus sign.

(-B) means (0.0 - B)

Exercises

51

Which arithmetic operation would be performed first in this expression?

A + B / C

ANSWER ON P. 173

52

Which arithmetic operation would be performed first in this expression?

(A + B) / C

ANSWER ON P. 175

53

Which in this expression?

(A + B) ** 2

ANSWER ON P. 175

56

What is the value of this expression?

8.0 / 4.0

ANSWER ON P. 172

55

What is the value of this expression?

(8.0 + 8.0)

ANSWER ON P. 173

54

What is the value of this expression?

8.0 + 8.0

ANSWER ON P. 171

57

What is the value of this expression?

7.0 + 8.0 / 4.0

ANSWER ON P. 165

58

What is the value of this expression?

8.0 / 4.0 + 8.0

ANSWER ON P. 168

59

What is the value of this expression?

(8.0 + 8.0) / 4.0

ANSWER ON P. 162

60

What is the value of this expression?

(A + B) / C if
A = 8.0
B = 8.0
C = 4.0

ANSWER ON P. 172

61

What is the value of this expression?

2.0 - (3.0 + 4.0)

ANSWER ON P. 168

62

What is the value of this expression?

2.0 - 3.0 + 4.0

ANSWER ON P. 169

63

Write the Fortran expression for the following algebraic formula:

$$a + b - \frac{cd^2}{e+c}$$

ANSWER ON P. 173

64

Now rewrite the expression using the following values for each of the variables:

A = 1.0 B = 2.0
C = 3.0 D = 4.0

E = 5.0

ANSWER ON P. 165

65

Now perform the operation indicated in the parentheses. Then remove the parentheses, and rewrite the formula, still using the numbers for the variables.

ANSWER ON P. 175

66

Now, performing the first-level operations in the Hierarcy of Operations, carry out the exponentiation, and rewrite the formula.

ANSWER ON P. 169

67

Now, working from the left, perform the first second-level operation, and rewrite the formula.

ANSWER ON P. 173

68

Now, again moving from the left, perform the other second-level operation and rewrite the formula.

ANSWER ON P. 175

69

Next, perform the left-most third-level operation and rewrite the formula.

ANSWER ON P. 169

70

Now perform the last remaining operation.

ANSWER ON P. 172

Mixed Expressions

Calculations are carried out in the computer in Fixed Point Mode or in Floating Point Mode.

Floating Point calculations are used for most arithmetic computations.

Fixed Point calculations are used primarily for counting and indexing operations.

A reminder:
Integer Mode (Fixed Point) variables *must* have

I J K L M or N

as the first character.
Floating Point Mode variables must begin with

A B C D E F G H Ø P Q R S T U V W X Y Z

In general, the calculations specified in one expression must be entirely of one mode, and must not be a mixture of modes.

For example,

A + I

is a *mixed* expression because "A" is in the Floating Point Mode,

and "I" is in the Fixed Point Mode.

Similarly,

J + 3.1416

is also a mixed expression because "J" is Fixed Point and 3.1416 is Floating Point.

Mixed Expressions are illegal, with one exception.

The only exception to the Mixed Expression rule is *exponentiation.*

Floating Point variables and constants *may be raised to Fixed Point* powers.

X ** 2 is legal
X ** K is legal

2 ** X is *not* legal
N ** 2.0 is *not* legal

The illegality of mixed expressions is not arbitrary. It follows from the different ways the numbers are stored in the computer.

Raising *whole* numbers to fractional powers is *not* reasonable because the result will be *fractional.*

Raising *fractional* numbers (Floating Point) to whole number (Fixed Point) powers is reasonable because the result will be *fractional*.

Exercises

71

What mode are the following Fortran Variables?

A B

ANSWER ON P. 173

72

What mode are the following Fortran Expressions?

A + B
A - C
A * C
A / C
A ** C

ANSWER ON P. 175

73

What mode are the following Fortran Variables?

K M

ANSWER ON P. 172

74

What mode are the following Fortran Expressions?

K + M
K - M
K * M
K / M
K ** M

ANSWER ON P. 173

75

Is the following expression legal?

X ** 2

ANSWER ON P. 172

76

What mode is

X ** 2

ANSWER ON P. 175

77

Is the following expression legal?

2 ** X

ANSWER ON P. 169

Part Six: ARITHMETIC STATEMENTS

General Form ● Incrementing ● Accumulating ● Defining a Variable ● Conversion ● Exercises

General Form

A Fortran Arithmetic *Statement* has the following general form:

Variable = Expression

After the numerical value of the expression is calculated, its value is stored as the value of the variable on the left of the equal sign.

A = B + C

is a Fortran Arithmetic Statement.

X = ALPHA / (C ** I)-FACTOR / SLIDE

is a Fortran Arithmetic Statement.

I = K + L

is a Fortran Arithmetic Statement.

The effect of an Arithmetic Statement (when executed) is first to perform all the operations in the expression to the *right* of the equal sign, reducing that expression to one numerical value.

Then this value is stored in the storage location for the variable named on the *left* of the equal sign.

Everytime a variable is set equal to the value of an expression,

Variable = Expression

the previous value of the variable is replaced by the new value. The old value (if any) is lost.

Incrementing

In Fortran,

INCREMENTING

may be accomplished by a statement like this:

```
I = I + 1
```

The effect of

```
I = I + 1
```

is to add one to the current value of I and then to store this sum back in I. In other words, it adds one to the current value of I.

Accumulating

In Fortran,

ACCUMULATING A TOTAL

may be accomplished by a statement like this:

```
SUMX = SUMX + X
```

The effect of

```
SUMX = SUMX + X
```

is to add the current value of X to the current value of SUMX and then to store the sum back in

SUMX.

Defining a Variable

Note that in all the examples cited so far, it has been assumed that all the variables in the expression had "current values."

For a variable to have a current value, it must have been used in some previous statement which, when executed, would have the effect of storing a value in its storage location.

Variables that have current values are said to be "defined."

Variables on the left side of the equal sign in arithmetic statements are defined because the numerical value of the expression will be stored in the variable.

Variables that appear in COMMON statements are also said to be defined, because their value presumably has been established in some previous Subroutine or in the Main Program.

Variables appearing on the Variable List of a READ statement are defined. Numerical values are read in for them.

The indexing variable in a DO statement (to be discussed later) is defined. The indexing variable is INITIALIZED and INCREMENTED each time the DO-loop is repeated.

Conversion

In all arithmetic statements cited so far, the variable on the left of the equal sign has been of the same mode as the expression. This *need not be* the case.

If a Floating Point variable is set equal to a Fixed Point expression,

the value of the Fixed Point expression is converted to Floating Point mode before it is stored in the Floating Point variable.

For example,

if N equals 4

FLN = N

will set FLN equal to 4.0

A Fixed Point variable may be set equal to a Floating Point expression.

In this case, the value of the Floating Point expression is "truncated." The fractional part of the value is discarded, and the Fixed Point variable is set equal to the whole part of the value of the Floating Point expression.

For example, if B equals 64.872

J = B

will set J equal to 64.

The entire fractional part is truncated.

78

The *general form* of an arithmetic statement in Fortran is

____ =_____.

ANSWER ON P. 162

79

Write an arithmetic statement that will set the variable DIST equal to the expression, TIME, multiplied by RATE.

ANSWER ON P. 172

80

Write an arithmetic statement that will *add one* to the variable I.

ANSWER ON P. 172

81

Write an arithmetic statement that will increase SUMX by the current value of X.

ANSWER ON P. 169

82

Write an arithmetic statement that will increase the value of SUMX by X(I).

ANSWER ON P. 173

83

What would be the value of X after execution of the following sequence of arithmetic statements?

```
. . .
A = 3.5
B = 4.5        X = A + B
               . . .
```

ANSWER ON P. 172

84

What would be the value of Y after execution of this sequence of arithmetic statements?

```
. . .          U = T * 2.0
R = 8.0        Y = U/S
S = 4.0        . . .
T = R + S
```

ANSWER ON P. 170

Part Seven: CONTROL STATEMENTS

Normal Flow of Control ● Unconditional Transfer of Control ● Conditional Transfer of Control ● IF-Loops ● DO-Loops ● Exercises ● A Sorting Example ● Sorting Task ● Diagnosis

Normal Flow of Control

The order in which statements are executed is extremely important in every program.

In Fortran programs, the Normal Flow of Control (the normal order of execution) is the actual sequence in which the statements appear in the program.

Normal Flow of Control means that when all the operations specified in one statement have been completed, control passes to the next statement forward in sequence.

Normal Flow of Control *does not* apply to non-executable statements. It does not apply to

DIMENSIØN, CØMMØN, FØRMAT, CØMMENT statements.

Whenever these kinds of statements appear in the path of control, they are skipped, i. e., control ignores them.

Normal Flow of Control applies to

Arithmetic Statements

READ Statements

PRINT Statements

There is, however, a group of Fortran Control statements that causes control to be transfered to a *specified* statement that may or may not be the next one in sequence.

There are three basic kinds of Control Statements:

Unconditional Transfer

Conditional Transfer

Loop-defining

Unconditional Transfer of Control

The general form of a simple unconditional transfer statement is

GØ TØ n

where "n" is the number of some other statement in the program.

When executed, the GO TO statement unconditionally transfers control to the statement number specified.

The GO TO statement is called an unconditional transfer because it *always* transfers control to one particular statement every time it is executed.

REMINDER

Any statement in a program may be numbered for reference purposes.

One reason for numbering a statement is so that there may be a GO TO statement that transfers control to it.

For example,

```
        . . .
        . . .
55      X = Y + Z / A ** 2
        . . .
        . . .
        . . .
74      GØ TØ 55
        . . .
```

When control reaches statement 74, the GO TO 55 causes it to transfer to statement 55.

Conditional Transfer of Control

The *conditional* transfer statement in Fortran transfers control to one of three statements depending on the value of a specified Fortran expression.

The basic conditional transfer statement in Fortran is an IF statement.

The general form of an IF statement is

IF (e) n_1, n_2, n_3

where "e" is *any* arithmetic expression, and where n_1, n_2, and n_3 are statement numbers.

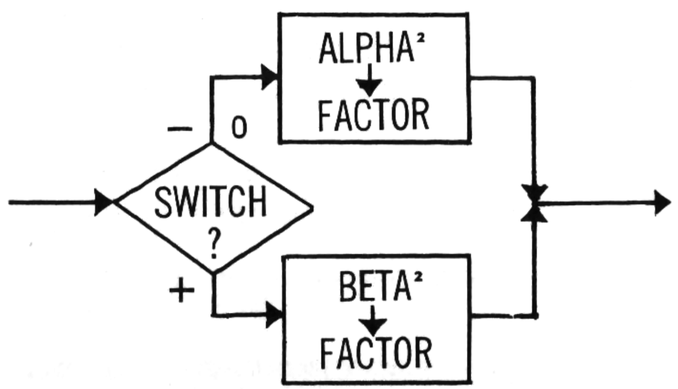
Control is transfered to n_1 if "e" is negative.
Control is transfered to n_2 if "e" is zero.
Control is transfered to n_3 if "e" is positive.

The branches of an IF statement may all be forward in the program; they may all be backwards; or they may be a combination of the two types of branching.

An IF statement, then, is like a three-way fork. The Flow of Control is to one of three statements (specified in the IF statement). These statements may be anywhere in the program.

For example, if you want to compute the value of FACTOR using ALPHA on some occasions, and using BETA on other occasions, you could branch control to either of the two paths depending on the value of SWITCH.

In a Flow Diagram this problem would look like this:



In the program the problem would look like this:

```
        . . .
        IF (SWITCH) 10, 10, 20
10      FACTOR = ALPHA ** 2
        GØ TØ 30
20      FACTOR = BETA ** 2
30      . . .
```

If the value of SWITCH is negative or zero, control will pass from the IF statement to statement 10. FACTOR will be computed using ALPHA. Control will pass (normally) to the GO TO statement. And then to statement 30, bypassing statement 20.

If the value of SWITCH is positive, control will pass from the IF statement to statement 20, bypassing statement 10 and the GO TO statement. FACTOR will be computed using BETA, and control will pass (normally) to statement 30.

IF-Loops

An IF statement may also branch control backwards in a program.

The repeated branching of control backwards by an IF statement is called an IF-loop.

If you wanted to repeat a series of statements *ten times* (i. e., construct a looping process), you would need three basic things. A *Counter*, a *Tester*, and an *Incrementer*.

The Counter to record the number of times the series had been repeated,

the Tester to keep control cycling within the loop until the tenth pass was completed then to transfer control to the next executable statement outside the loop,

the Incrementer to increase the Counter by one at each pass in the loop.

To accomplish this in a program, you could define a variable to be used as the counter; use an IF statement to

repeat the loop the prescribed number of times; and write a special arithmetic statement that would increment the counter by one at each pass:

The IF statement would fulfill the three following functions:

1. Testing the counter variable at each pass for equality with ten.

2.

Branching control at each pass to the incrementing statement, which would increase the counter variable *by one* before going back to the first statement in the loop.

3.

Taking control out of the loop after the tenth pass test and branching it to the next executable statement in the program.

The form could be like this:

```
        . . .
        I = 1
45      . . .
        . . .
        IF (10 - I) 56, 56, 75
75      I = I + 1
        GØ TØ 45
56      . . .
```

As long as "I" is less than ten, the expression in the IF statement will be positive; and control will branch to statement 75 where "I" will be incremented by one. Then control will be transferred back to statement 45.

As soon as (10 - I) is zero, control will branch to statement 56.

DO-Loops

An easier way to set up loops is with a DO statement.

A DO statement may be put at the beginning of the series of statements to be repeated.

A DO statement differs from an IF statement in the following manner:

It specifies the counting (or index) variable, gives its initial and final value, and indicates the statement number of the last statement in the series.

The general form of a DO statement is

```
DØ n   i = m1, m2
   . . .
   . . .   other statements
   . . .   in the series
n  . . .
```

"n" is the statement number of the last statement in the series of statements to be repeated.

"i" is the index variable.

"m_1" is the initial value of the index variable.

"m_2" is the final value of the index variable.

"n" must be a statement number.

"i" must be a Fixed Point variable

"m_1" must be a Fixed Point constant or a Fixed Point variable.

"m_2" must be a Fixed Point constant or a Fixed Point variable.

For example,

```
        . . .
        DØ 88  I = 1, 10
        . . .
        . . .
        . . .
88      CØNTINUE
        . . .
```

All the statements following the DO statement will be executed in sequence until control reaches statement number 88.

At this point, control will loop back to the first statement after the DO statement.

After the sequence has been repeated ten times, control will pass to the next statement following the CONTINUE statement.

The CONTINUE statement is a dummy statement that has no effect on the program.

It functions only as a convenient marker for the last statement in the range (the complete series of statements) of the DO statement.

The general form of a CONTINUE statement is

n CØNTINUE

where "n" is the statement number specified in the DO statement.

The "range" of a DO statement or DO-loop is the sequence of statements included in it.

There is no limit on the number of statements that may be in the range of a DO statement.

The first statement in the range of a DO-loop must be *executable*. It must *not*, for example, be a FORMAT statement.

The last statement in the range must *not* be a transfer statement. It must not, for example, be a GO TO or an IF statement.

You may have a statement in a DO-loop that transfers control out of the middle of the loop, but you may not have a statement in your program that would transfer control into the middle of the loop.

You may have CALL statements in a DO-loop.

And you may have DO-loops within DO-loops.

You may have IF statements in a DO-loop.

Loops within loops are executed as follows:

When control reaches the inner loop on the first pass, it remains there until that loop is satisfied.

Control then returns to the remaining statements in the outer loop. It remains there until it reaches the last statement.

Having completed the first pass, control then returns to the beginning of the outer loop and starts the entire process over again.

The DO statement governing the outer loop determines the number of times that control will reach the inner loop. Thus, an index variable (in a DO statement governing an outer loop) with parameters from one to one hundred, would cause control to reach the inner loop one hundred times.

The general structure of "nested" loops is

```
        . . .
        DØ 99  I = 1, 100
        . . .
        DØ 53  J = 1, 50
        . . .
        . . .
53      CØNTINUE
        . . .
99      CØNTINUE
        . . .
```

inner loop

outer loop

In this example each statement in the inner loop will be executed 5,000 times.

(Every time the inner loop is executed, each statement in it will be executed 50 times. And since the inner loop will be executed a total of 100 times, according to the example, each inner loop statement will, therefore, be executed a total of 50 × 100 times.)

Each time control reaches the inner loop, the index variable there is initialized causing control to remain in the loop until the the loop is satisfied.

Note:

All the statements in the range of the inner loop are within the range of the outer loop.

Both the inner and outer loop index variables may be used as subscripts in the inner loop; however, the inner loop index may not be used as an outer loop subscript.

Exercises

85

When executed , the Control statement GØ TØ 55 will cause control to be transferred to statement number_____.

ANSWER ON P. 166

86

The general form of a GO TO statement is GØ TØ n where "n" is the statement____ of an executable statement.

ANSWER ON P. 166

87

When executed, the Control statement

IF (X) 10, 20, 30

will cause control to be transferred to statement number 10 if X is _____.
To statement number 20 if X is _____.
And to statement number 30 if X is _____.

ANSWER ON P. 169

88

In the following group of statements,

```
66      X = X + XINC
        IF (X - XTEST) 66, 77, 77
77      . . .
        . . .
```

control will branch backwards to statement 66 until the expression (X - XTEST) is either _____ or _____.

ANSWER ON P. 166

89

In the following group of statements

```
If (WHICH) 100, 200, 300
100 . . .
    . . .
    GØ TØ 999
200 . . .
    . . .
    GØ TØ 999
300 . . .
    . . .
999 . . .
```

Control will branch forward to statement 100, if WHICH is _____.

Control will branch forward to statement 200, if WHICH is _____.

Control will branch forward to statement 300, if WHICH is _____.

Regardless of which path control follows, it will always reach statement number _____.

ANSWER ON P. 163

90

If the value of ABLE is larger than the value of BAKER, what would be the value of this expression (negative, zero, positive)?

(ABLE - BAKER)

ANSWER ON P. 169

91

When the following statement is executed,

IF (ABLE - BAKER) 10, 20, 30

where will control be transfered if ABLE = 10 and BAKER = 5?

ANSWER ON P. 165

92

If, when executed, this statement branches control to statement 60

IF (GAMMA - TEST) 40, 50, 60 we may deduce that the value of (GAMMA -TEST) was_____, and that GAMMA was_____ than TEST.

ANSWER ON P. 170

93

At the end of this series of statements,

```
IF (CHECK - TABLE) 10, 20, 20
10 BIG = TABLE
   GØ TØ 40
20 BIG = CHECK
   GØ TØ 40
40 . . .
```

BIG will equal TABLE if TABLE was _____ than CHECK.

ANSWER ON P. 165

94

BIG will equal CHECK if CHECK was _____ than TABLE, or if CHECK was _____ to TABLE.

ANSWER ON P. 170

95

The following DO-loop

```
DØ 666 I = 1, 1000
. . .
. . .
. . .
666 CØNTINUE
```

will be repeated exactly _____ times.

ANSWER ON P. 169

96

What will be the value of CRS (10) after this statement?

```
CRS(10) = M
```

if M = 64

ANSWER ON P. 163

97

What would be the value of Z(3) after execution of this sequence of statements?

```
      DIMENSIØN
      DØ 10 I = 1, 300
      Z(I) = I ** 2
10    CØNTINUE
      . . .
      . . .
```

ANSWER ON P. 163

98

In the following DO-loop

```
      DØ 74 K = 1, N
      Z (K) = 1.0
74    CØNTINUE
```

the first N entries in the list Z will all be set equal to _____.

ANSWER ON P. 170

99

In the following DO-loop

```
        SUMX = 0.0
        DØ 666 I = 1, 1000
        SUMX = SUMX + X(I)
666     CØNTINUE
        . . .
```

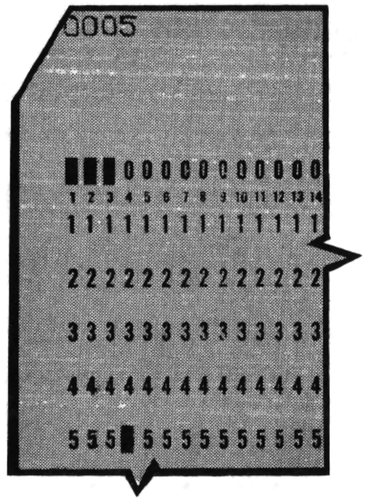
the total of all the values of X from X(1) to X(1000) will be stored in _____.

ANSWER ON P. 170

100

What would be the value of Z(N) after execution of this sequence?

```
        . . .
1       FØRMAT (I4)
        READ 1, N
        DØ 22 I = 1, N
        Z(I) = I ** 2
22      CØNTINUE
        . . .
```



ANSWER ON P. 163

101

Fill in the statement numbers in the IF statement in this DO-loop so that BIGE will equal the largest value of E when the loop is satisfied.

```
        BIGE = 0.0
        DØ 10 I = 1,N
        IF (BIGE - E(I))_______,_______,_______
77      BIGE = E(I)
10      CØNTINUE
```

ANSWER ON P. 170

102

What will be the significance of K after this loop has been executed?

```
        BIGT = 0.0
        DØ 99 M = 1, MEND
        IF (BIGT - T(M)) 54, 999, 999
54      BIGT = T(M)
        K = M
999     CØNTINUE
```

ANSWER ON P. 166

103

Write a DO-loop that will increase by 1.0 each value of X from X(1) to X(100).

ANSWER ON P. 166

104

Write another DO-loop that will increase all N values of Y by ten.

ANSWER ON P. 165

A Sorting Example

Numerical sorting is the systematic rearranging of a list of numbers in some prescribed order.

There are, of course, many ways to rearrange lists of numbers and many different ways to sort them.

Most ways, however, involve a repetitive process, and repetitive processes can usually be programmed as loops.

For example, if you were told to sort a list of numbers in descending order, you could proceed as follows:

You could repeatedly

(1) search through the list for the biggest number,

(2) copy that number onto a new list, and

(3) erase that number from the original list.

The detailed step-by-step process would be something like this:

Begin by assuming that the first number on the list was the biggest one. Then, by comparing it with the next number, the one following that one and so on, you would change your assumption each time a bigger one was found.

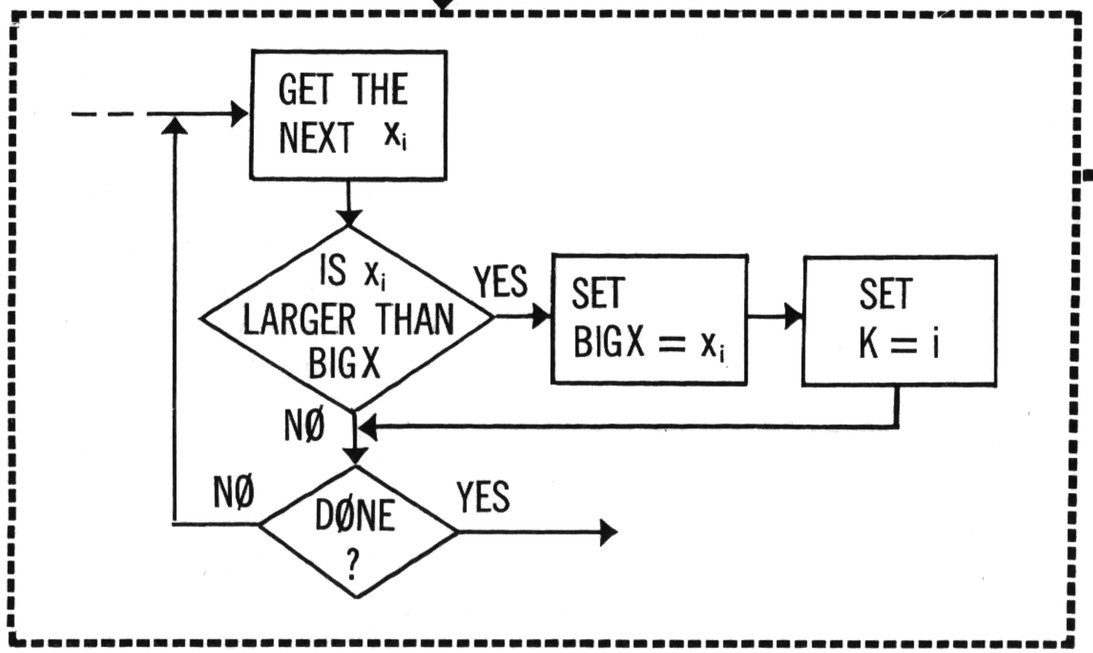
Keep track of the assumed biggest number by writing it down and noting its position on the list. Then every time you find a bigger number, repeat this process.

When you reach the bottom of the list the first time, you will have the first largest number. This number should be copied onto a new list. It will be the first number on this new list.

Before searching for the second biggest number on the original list, you must erase the first biggest one. The second time you search the list, you will find the second biggest number and copy it on the new list in the second position.

To arrange this sorting procedure so that it could be programmed in Fortran, you could represent the list as a subscripted variable. Each subscript would then represent a different number on the list.

The search and compare process may be illustrated in a flow diagram as follows:



Notice that there are alternate paths through loop:

1. If "x_i" *is not* bigger than "BIGX" control drops down to the "Done?" box, but
2. If "x_i" *is* bigger than "BIGX", the value of "x_i" replaces "BIGX" and "k" is set equal to "i."

"BIGX" is the currently assumed biggest number on the list.

"k" is its position on the list. When there is no change in this assumption, the no path is followed. When the assumption is changed, the yes path is followed.

The Fortran program to carry out this process would consist of two loops: An *inner loop* to find each biggest number and its position on the list, and an *outer loop* to move that number to the next position on a new list and to erase it from the original list.

The Flow Diagram on the following page is a description of an entire sorting process.

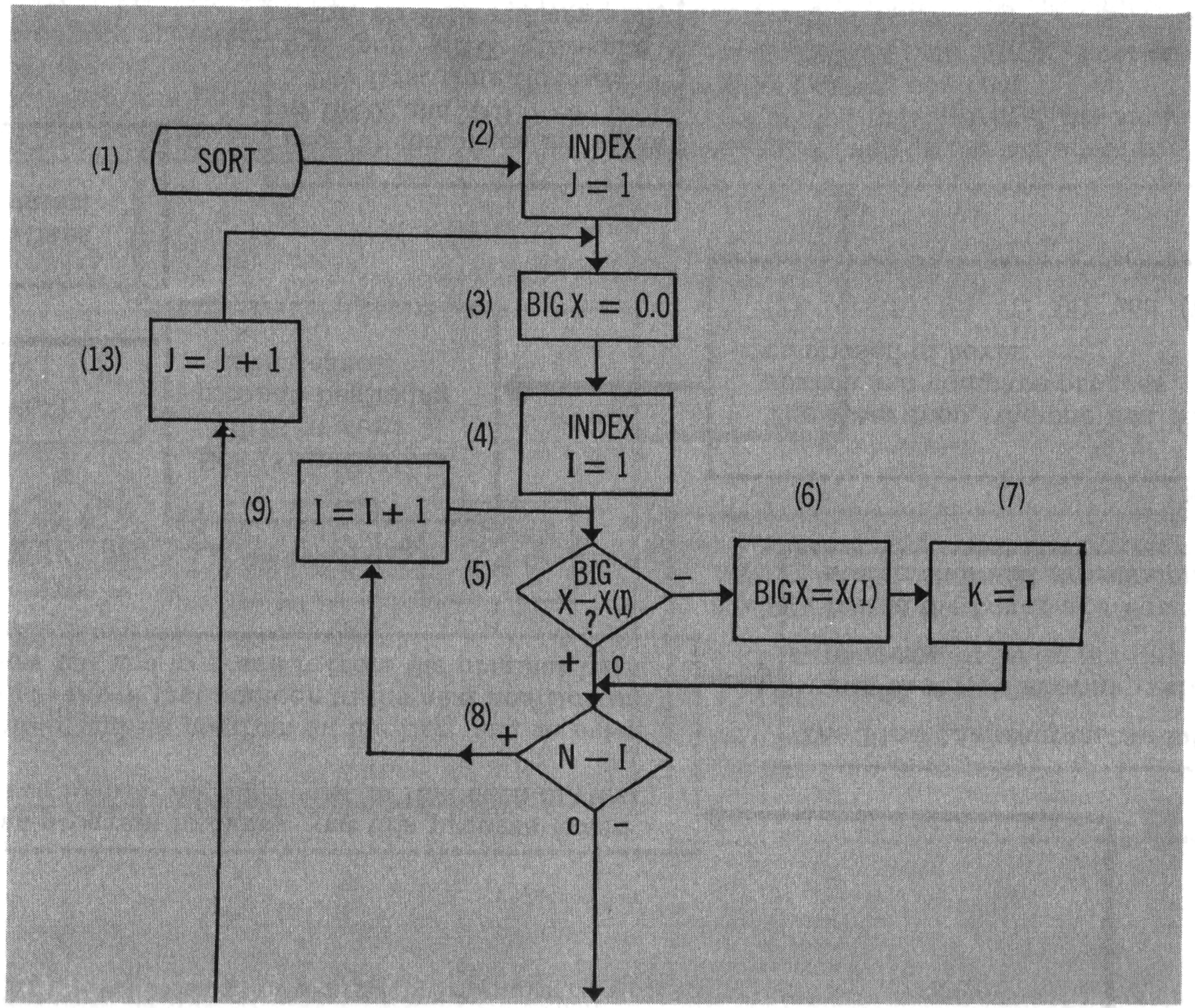
Inside the boxes are Fortran-like statements and expressions.

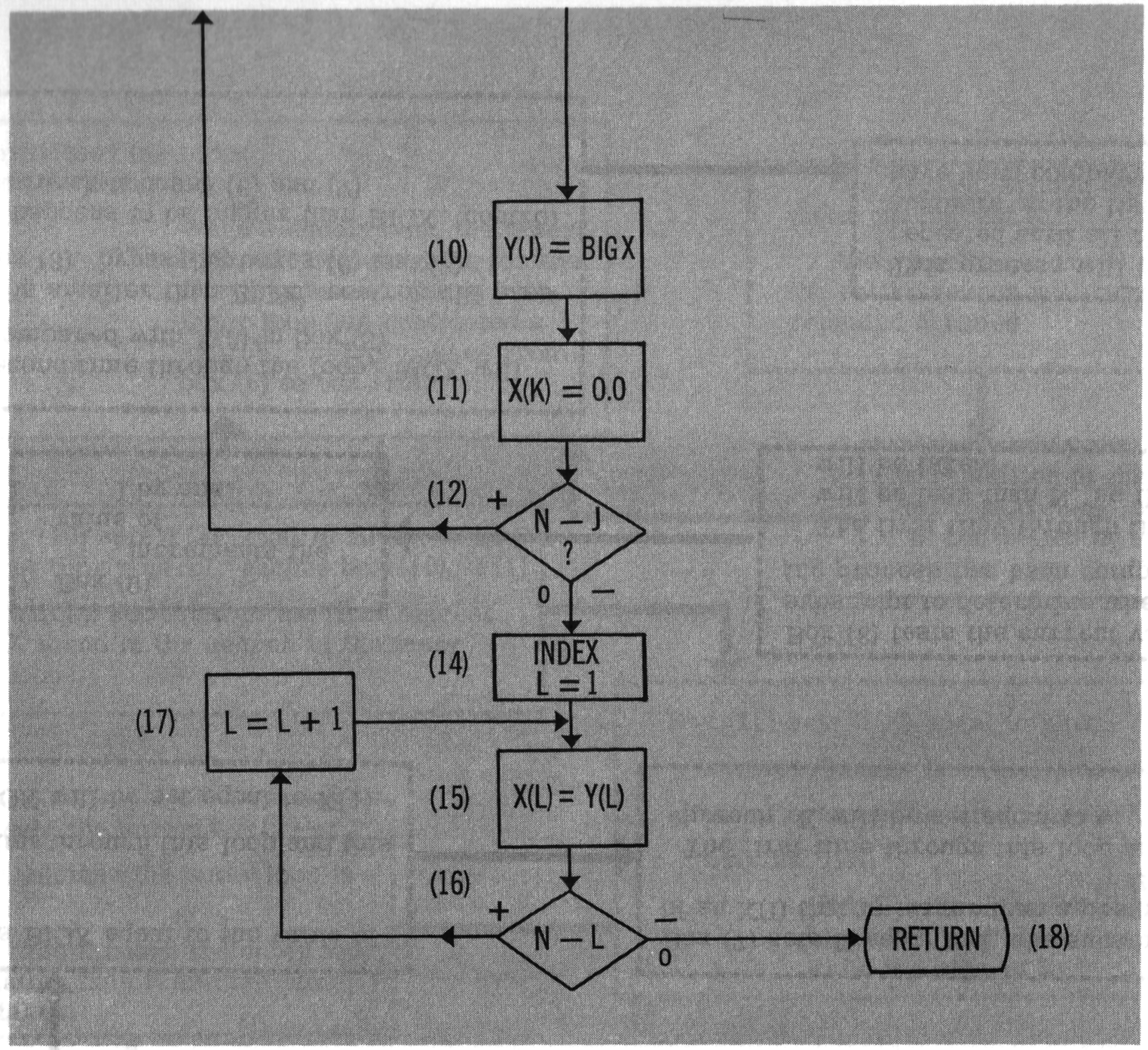
The inner loop, the one that does the search and compare process, is composed of boxes (3) (4) (5) (6) (7) (8) and (9).

Box (3) initializes BIGX at zero prior to beginning each search.

Box (4) initializes the index counter.

Box (5) compares the value of BIGX and X(I).
The first time through this loop, BIGX will equal zero, so X(1) will certainly be bigger, and the minus exit will be taken.


(1)
SORT
(2)
INDEX
J = 1
(3)
BIG X = 0.0
(13)
J = J + 1
(4)
INDEX
I = 1
(9)
I = I + 1
(6)
(7)
(5)
BIG
X — X(I)
?
–
BIGX=X(I)
K = I
+
0
(8)
+
N – I
0
–


(10)
Y(J) = BIGX
(11)
X(K) = 0.0
(12)
+
N — J
?
0
—
(14)
INDEX
L = 1
(17)
L = L + 1
(15)
X(L) = Y(L)
(16)
+
N — L
—
0
RETURN
(18)

Box (6) sets BIGX equal to the value of X(I).

The first time through this loop and this branch, BIGX will be set equal to X(1).

Box (7) sets K equal to I, the subscript of an X(I) that is larger than a previous BIGX.

The first time through this loop and branch, K will be set equal to 1.

Box (8) tests the current value of the subscript to determine whether the looping process has been completed.

The first time through this loop, I will be less than N, so the plus exit will be taken.

Box (9) increments the value of I by one.

The second time through the loop, BIGX will be compared with X(2) in box (5).

If X(2) is smaller than BIGX, control will pass to box (8), bypassing boxes (6) and (7).

If X(2) happens to be bigger than BIGX, control will branch to boxes (6) and (7).

This process will be repeated until all N numbers on the list have been compared.

Box (2) initializes the index counter for the outer loop.

When J equals one, the inner loop will be searching for the first biggest X. When the inner loop has completed a search, control passes from box (8) to box (10).

One search through the list will yield *one* biggest X.

And since there are N values of X, the entire search will have to be repeated N times.

The repetition of the entire searching process is controlled by the outer loop.

Box (10) sets Y(J) equal to BIGX. The first time control reaches box (10), Y(1) will be set equal to the first biggest X found in the search in the inner loop.

Box (11) sets X(K) equal to zero. The first time control reaches box (11), this will have the effect of erasing the first biggest X from the original list.

(If this step is omitted, the first biggest X will always turn out to be the biggest X.)

Box (12) tests the current value of J to determine whether the outer loop is finished.

The first time through the outer loop, J will be smaller than N and the plus exit will be taken.

Box (13) increments the value of J by one.
The second time through the loop, BIGX and the I index counter must be reinitialized.
If BIGX is not reinitialized, it will equal the first biggest X, and the second biggest X will not be found. If I is not reinitialized, the search will not begin with the first X.

The outer loop will reinitialize and execute the inner loop until N biggest X's have been transfered to Y. When the outer loop is satisfied, Y will contain all the original numbers sorted in descending order.

In box (12), J will equal N, causing control to pass to box (14).

Box (14) initializes another loop which is designed to transfer the sorted list in Y back into X.

The first time control reaches box (14), L will be set equal to one.

Box (15) sets X(L) equal to Y(L).

The first time control reaches this box, X(1) will be set equal to Y(1).

Box (16) tests the index counter to determine whether this loop is finished.

The first time control reaches this box, L will be less than N so the plus exit will be taken.

Box (17) increments the value of L by one.

The second time through this loop, L is equal to two. X(2) is set equal to Y(2).

This transfer process is continued until all N numbers in Y have been moved back into X.

When all N values have been transferred, L will equal N, and control will follow the minus-zero exit from box (16), and the work of the routine is completed.

Box (18) returns control to the Main Program.

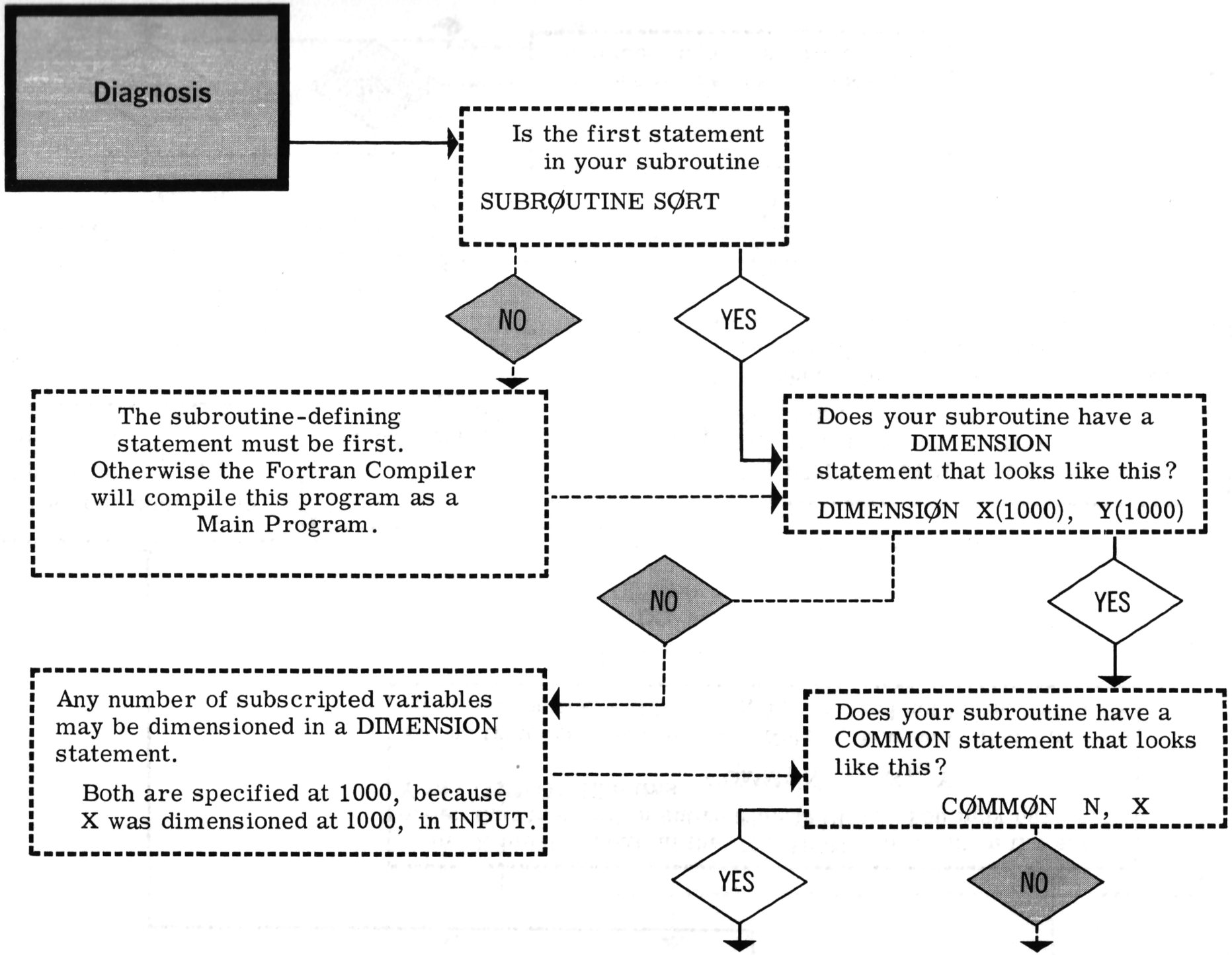
Box (1) identifies the routine as SUBRØUTINE SØRT

Sorting Task

Using the Flow Diagram on page 134 as a model, write a SORT subroutine that will sort the one-dimensional array "X" in descending order. Remember that N and X are in Common Storage, don't forget to dimension both X and Y.

There are three loops in the Flow Diagram. You may use DO statements to set up these loops, or you may use IF statements. DO-loops, however, are generally more efficient, and are, therefore, the recommended technique.

Remember that you must reserve 1,000 storage locations for X in Common Storage. Remember that N is in Common Storage.


Diagnosis
Is the first statement in your subroutine
SUBRØUTINE SØRT
NO
YES
The subroutine-defining statement must be first. Otherwise the Fortran Compiler will compile this program as a Main Program.
Does your subroutine have a DIMENSION statement that looks like this?
DIMENSIØN X(1000), Y(1000)
NO
YES
Any number of subscripted variables may be dimensioned in a DIMENSION statement.
Both are specified at 1000, because X was dimensioned at 1000, in INPUT.
Does your subroutine have a COMMON statement that looks like this?
CØMMØN N, X
YES
NO

N and X must appear in the COMMON statement in the same order that they did in subroutine INPUT. You may have put Y in COMMON like this

```
CØMMØN N, X, Y
```

and no troubles would ensue. Notice, however, that since no other subroutine will be using Y, it need not be in COMMON.

Does your subroutine have a RETURN statement?

NO

The RETURN statement is absolutely essential to return control to the Main Program.

It should be the last statement to be executed in the subroutine.

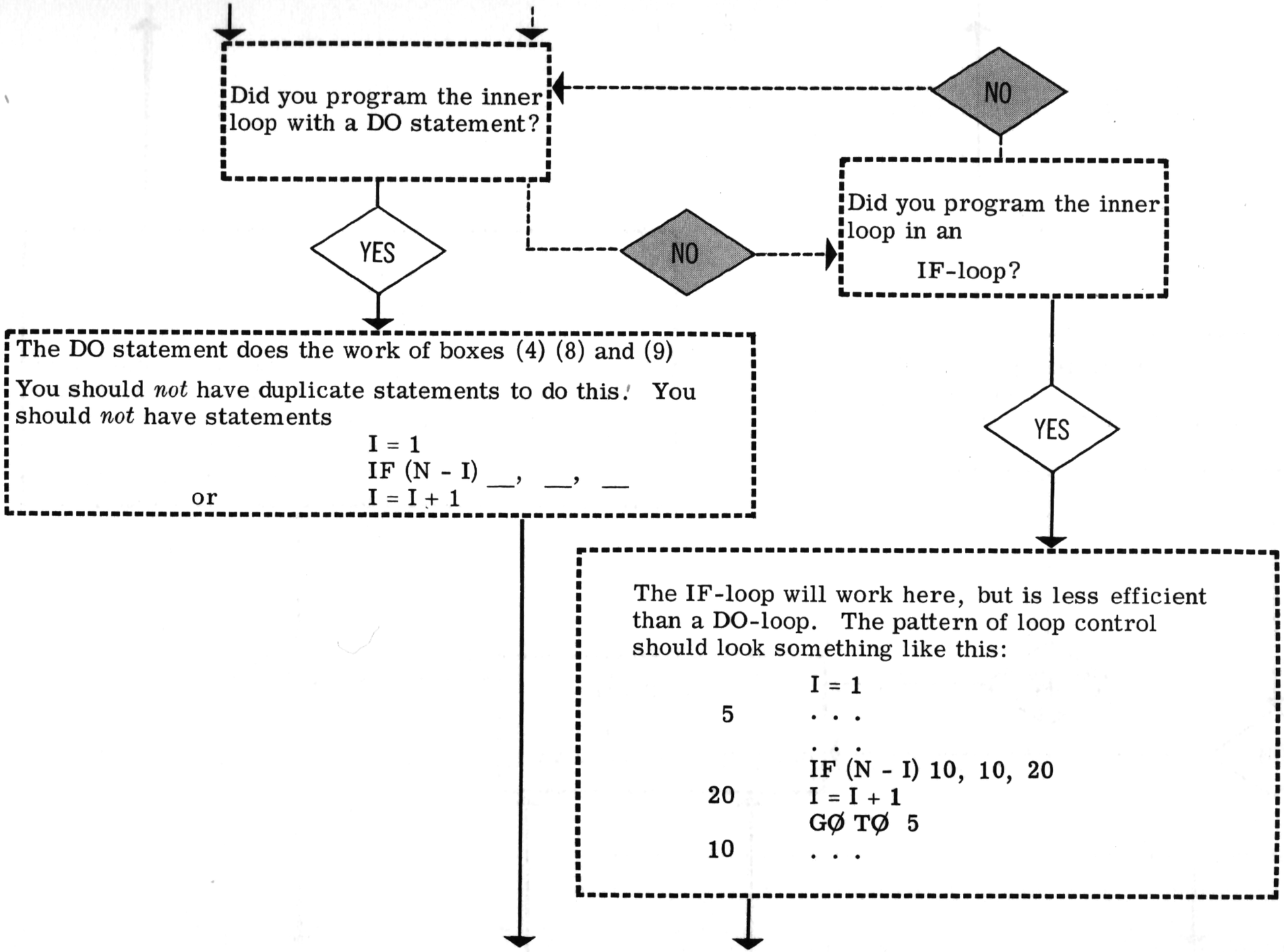
YES

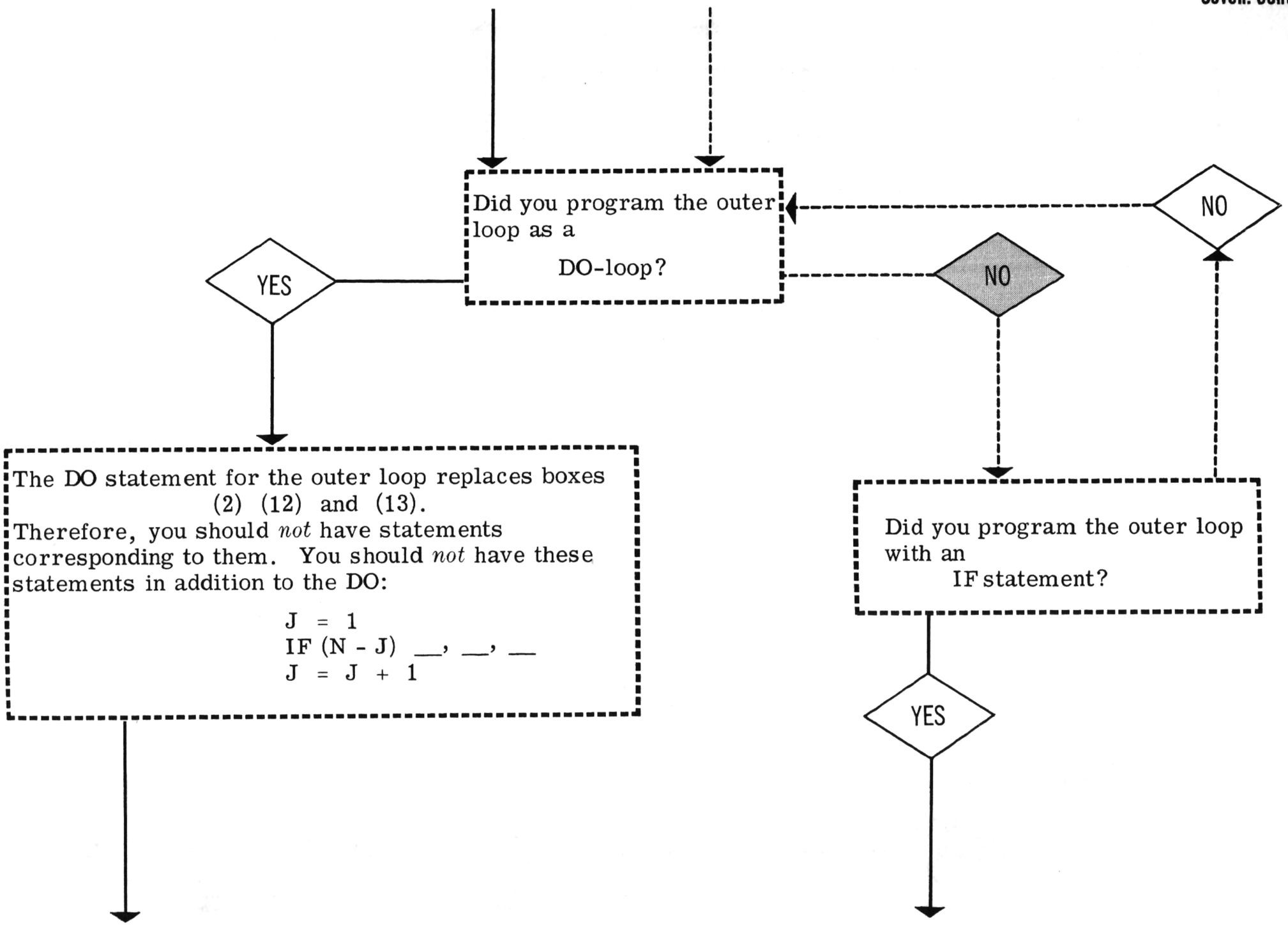
Is the very last statement in your subroutine an END statement?

YES

NO

The END statement is also absolutely essential. It marks the end of the statements to be included in your subroutine.


Did you program the inner loop with a DO statement?
NO
Did you program the inner loop in an IF-loop?
YES
NO
The DO statement does the work of boxes (4) (8) and (9)
You should *not* have duplicate statements to do this. You should *not* have statements
I = 1
IF (N - I) __, __, __
or
I = I + 1
YES
The IF-loop will work here, but is less efficient than a DO-loop. The pattern of loop control should look something like this:
I = 1
5 . . .
. . .
IF (N - I) 10, 10, 20
20 I = I + 1
GØ TØ 5
10 . . .


Did you program the outer loop as a DO-loop?
YES
NO
NO
The DO statement for the outer loop replaces boxes (2) (12) and (13).
Therefore, you should *not* have statements corresponding to them. You should *not* have these statements in addition to the DO:
J = 1
IF (N - J) __, __, __
J = J + 1
Did you program the outer loop with an IF statement?
YES

The pattern of control for the IF-loop should look something like this:

```
        J = 1
77      . . .
        (inner loop)
        . . .
        IF (N - J)  66,  66,  55
55      J = J + 1
        GØ TØ  77
66      . . .
```

Did you program the transfer back loop as a DO-loop?

YES

NO

NO

Did you program the transfer back loop as an IF statement?

YES

The DO statement for this loop replaces boxes (14) (16) and (17)
Therefore, you should not have statements corresponding to them. You should not have

```
L = 1
IF (N - L) __,  __,  __
L = L + 1
```

The pattern of control for this IF-loop should look something like this:

```
        L = 1
43      . . .
        IF (N - L)  27,  27,  99
99      L = L + 1
        GØ TØ  43
27      . . .
```

If necessary, make a clean and correct copy of your program.

Then go on to the next section.

Part Eight: OUTPUT STATEMENTS

Relation to Input ● Output Field Specifications ● Carriage Control ● Relations between Fields, Specifications, and the Variable List ● Exercises ● Output Task ● Diagnosis ● Conclusion

Relation to Input

Planning output is the

reverse

of planning input.

Input FORMAT statements *describe* how numbers are already arranged on data cards.

Output FORMAT statements *specify* how numbers are to be arranged on a printed page of output.

READ statements specify where the numbers shall be stored.

PRINT statements specify which numbers, already in storage, shall be printed out.

In the output process there are two statements involved:

A FORMAT statement

and

a PRINT statement

Output Field Specifications

The first step in planning output is designing the layout of each printed line.

Each printed line is comprised of 120 "print positions."

Output FORMAT specifications *must* account for every print position beginning at the left. Extra positions at the right may be omitted.

There are four kinds of fields that may be specified in an output FORMAT statement:

- I - for Fixed Point
- F - for Floating Point
- X - for Blanks
- H - for Hollerith characters

"X" and "H" specifications are used for spacing and labeling.

"I" and "F" are used to print the values of the variables in the Variable List in the PRINT statement.

If the variables are Fixed Point mode, they should be printed with the

I

FORMAT statement specification.

The general form for the "I" specification is

Iw

where "w" is the number of print positions in the field (i.e., the width).

If the value of a variable were 87 and if it were printed with an I4 format, it would look like this:

⊙⊙87

(The blanks "⊙⊙" would not be printed.)

If the variable is Floating Point, its value should be printed with an

F

FORMAT statement specification.

The general form of the "F" specification is

Fw.d

"w" is the number of print positions of the field.
".d" is the number of digits to be printed in the fractional part of the value.

If the value of a variable were 95.273842 and if it were printed with an F5.2 format specification, it would look like this:

95.27

Note that the fractional part is rounded to two places, which is the number of places (digits) specified in the F5.2 format specification.

The general form of the X FORMAT specification is

nX

where "n" is the number of the print positions to be left blank.

The general form of the H FORMAT specification is

$n\,H\,c_1\,c_2\,c_3 \ldots c_n$

"n" is the number of characters to be printed.

$c_1\,c_2\,c_3 \ldots c_n$ are the actual characters.

They may be alphabetic, numeric, or special characters.

Note:
"n" nust be precise because the next n characters in the FORMAT statement, following the "H," will be printed.

For example, if

23HTHIS⊙IS⊙THE⊙FIRST⊙PHASE

were specified in a FORMAT statement, the printed output would look like this:

THIS IS THE FIRST PHASE

"23H" indicates that there will be 23 Hollerith characters in this field of output, and the dotted zeros indicate the character positions that will be left blank.

The four kinds of output specifications (I, F, X, and H), used individually or in combination, specify the layout of one line of printing.

Carriage Control

The position of a line on a page is specified by a "carriage control" character. This character is and must be the first character specified by the FORMAT statement.

The format specification for the first print position is the "carriage control" character.

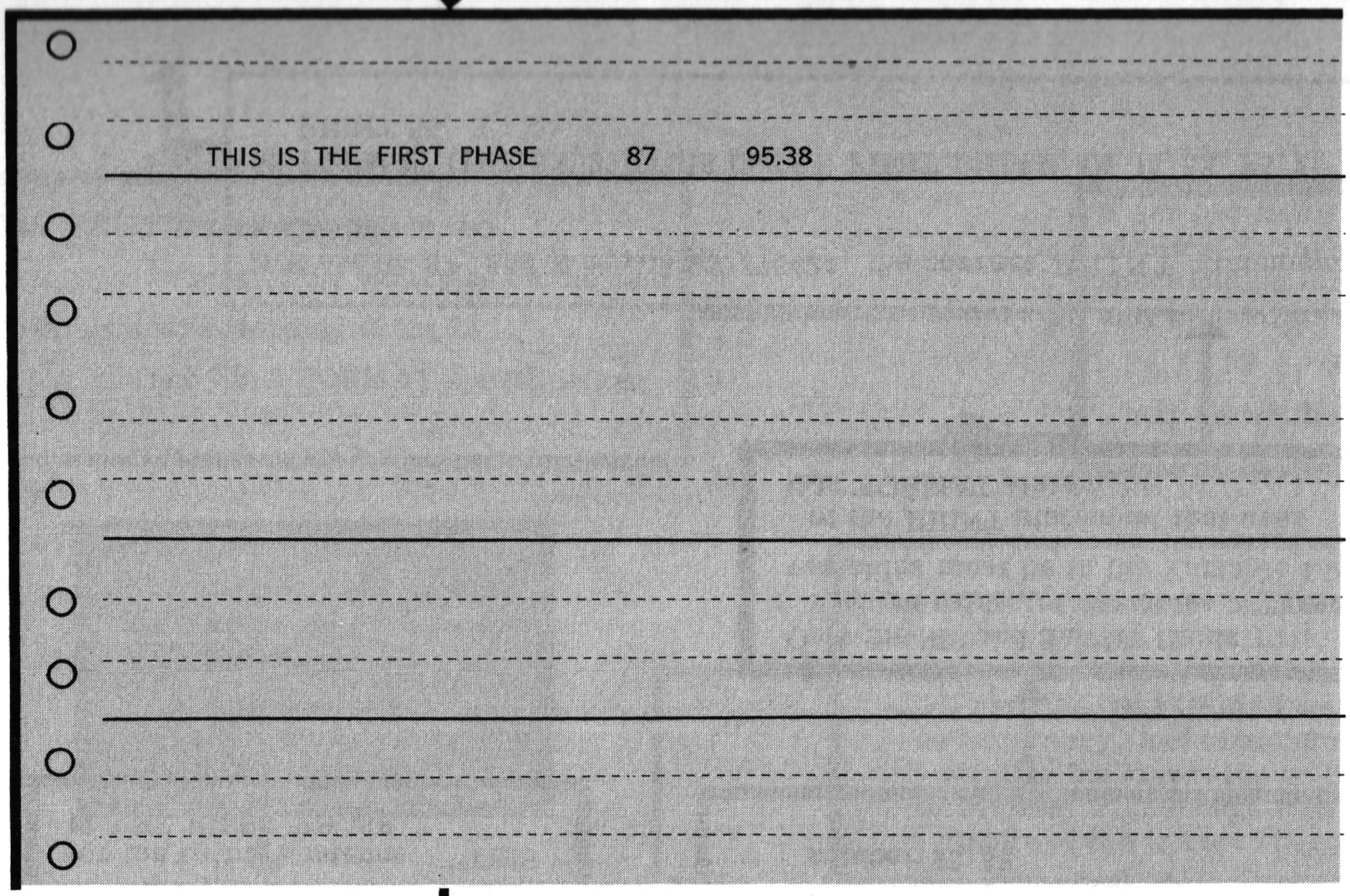
The carriage control character is *not* printed.
It is an instruction to the printer to space down before printing.

The following carriage control characters are interpreted by the printer as follows:

⊙	(blank)	space down one line before printing
0	(zero)	space down two lines before printing
1	(one)	skip to new page before printing

Carriage control characters are specified as Hollerith fields of one character:

1H⊙	specifies a blank as the first character in the line and instructs the printer to space down one line.
1H0	specifies a zero as the first character and instructs the printer to space down two lines
1H1	specifies a one as the first character and instructs the printer to skip to a new page



Assuming that this output is to be started on a new page, the FORMAT statement describing it would be:

```
FØRMAT (1H1,6X,23HTHISØISØTHEØFIRSTØPHASE, 5X, I4, 5X, F5.2)
```

The first Hollerith field, "1H1," is carriage control. It causes the printer to skip to a new page before printing.

The second Hollerith field is for the titling sentence, "THIS IS THE FIRST PHASE."

The "X" fields provide spacing between the other fields.

The "I" field is for the number 87, and the "F" field is for the number 95.38.

Only the "I" and the "F" fields require values of variables. These variables must be in the Variable List of the PRINT statement that uses this FORMAT statement.

If K equals 87, and X equals 95.378424, the correct PRINT statement would be:

```
55 FØRMAT (1H1, 6X, 23HTHIS IS THE FIRST PHASE, 5X, I4, 5X, F5.2)
   PRINT 55, K, X
```

The PRINT statement causes the values of the variables on the List to be printed according to the I4 and F5.2 FORMAT specifications.

If several fields have identical formats, the specification for one of them may be multiplied:

F8.3, F8.3, F8.3, F8.3,

may be written 4F8.3

Relations between Fields, Specifications, and the Variable List

Just as with input FORMAT statements, output FORMAT specifications will be re-used whenever there are more variables on the List than there are specifications in the FORMAT statement.

The PRINT statement continues to print values for every variable on its List.

It repeats the FORMAT as often as necessary, and terminates after printing the last value on the List. It terminates even though it may be in the middle of the format specifications.

Thus, 45 FØRMAT (1HO, 5F10.4)
PRINT 45, (X(I), I = 1, 50)

would print five values of X on each line. There would be ten such lines, double spaced.

Exercises

105

Write a FORMAT statement with single spacing, a field of *eight* blanks, and a field of *ten* print positions for a Floating Point value retaining *four* digits in the fractional part. Number it 55.

ANSWER ON P. 163

106

Now write a Print statement that will print the value of the variable, "TEST," using FORMAT statement 55.

ANSWER ON P. 166

107

Write another PRINT statement that also uses FORMAT statement number 55, but this time to print out all 25 values of "Z."

ANSWER ON P. 166

108

How many values of "Z" will be printed on each line?

ANSWER ON P. 166

109

How many lines will be needed to print all 25 values of "Z"?

ANSWER ON P. 163

110

Write a FORMAT statement and a PRINT statement for 80 values of "Y" according to the following format specifications:

1. Eight values per line.
2. Each field nine print positions wide, with four digits in the fractional part.
3. Double space the lines.

ANSWER ON P. 170

111

Write a FORMAT and PRINT statement that will do nothing more than cause the printer to skip to a new page.

ANSWER ON P. 163

112

Write your own FORMAT and PRINT statement according to your own specifications. Then, on a piece of scratch paper, draw your conception of how the printed output will look.

ANSWER ON P. 174

Output Task

Write a subroutine called OUTPUT. Design it so that it will print all the values of X that were sorted in subroutine SORT. Plan the printed page so that it looks something like this:

TABLES OF VALUES SORTED IN DESCENDING ORDER

xx.xx

xx.xx

xx.xx

xx.xx

xx.xx

xx.xx

xx.xx

.

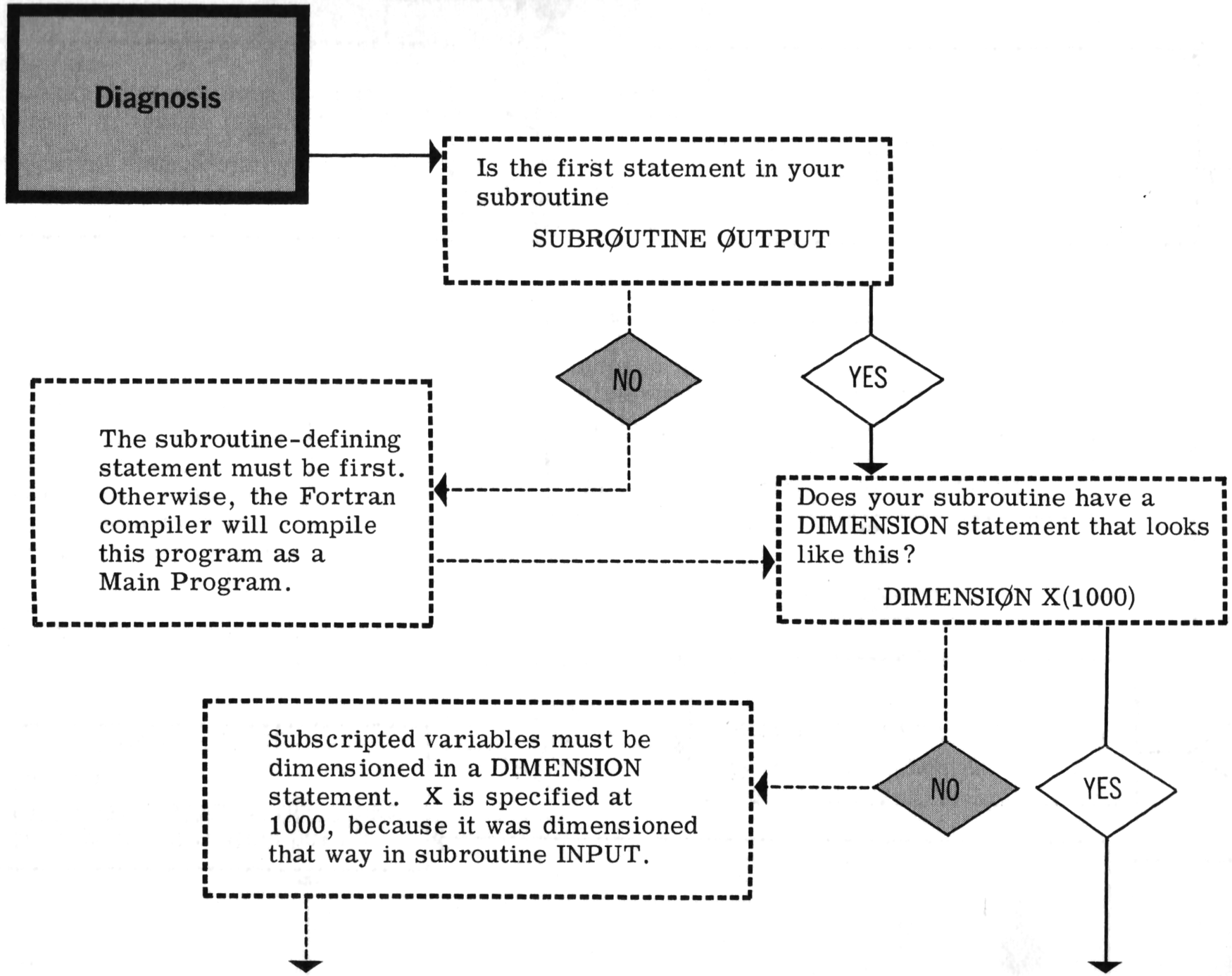
.

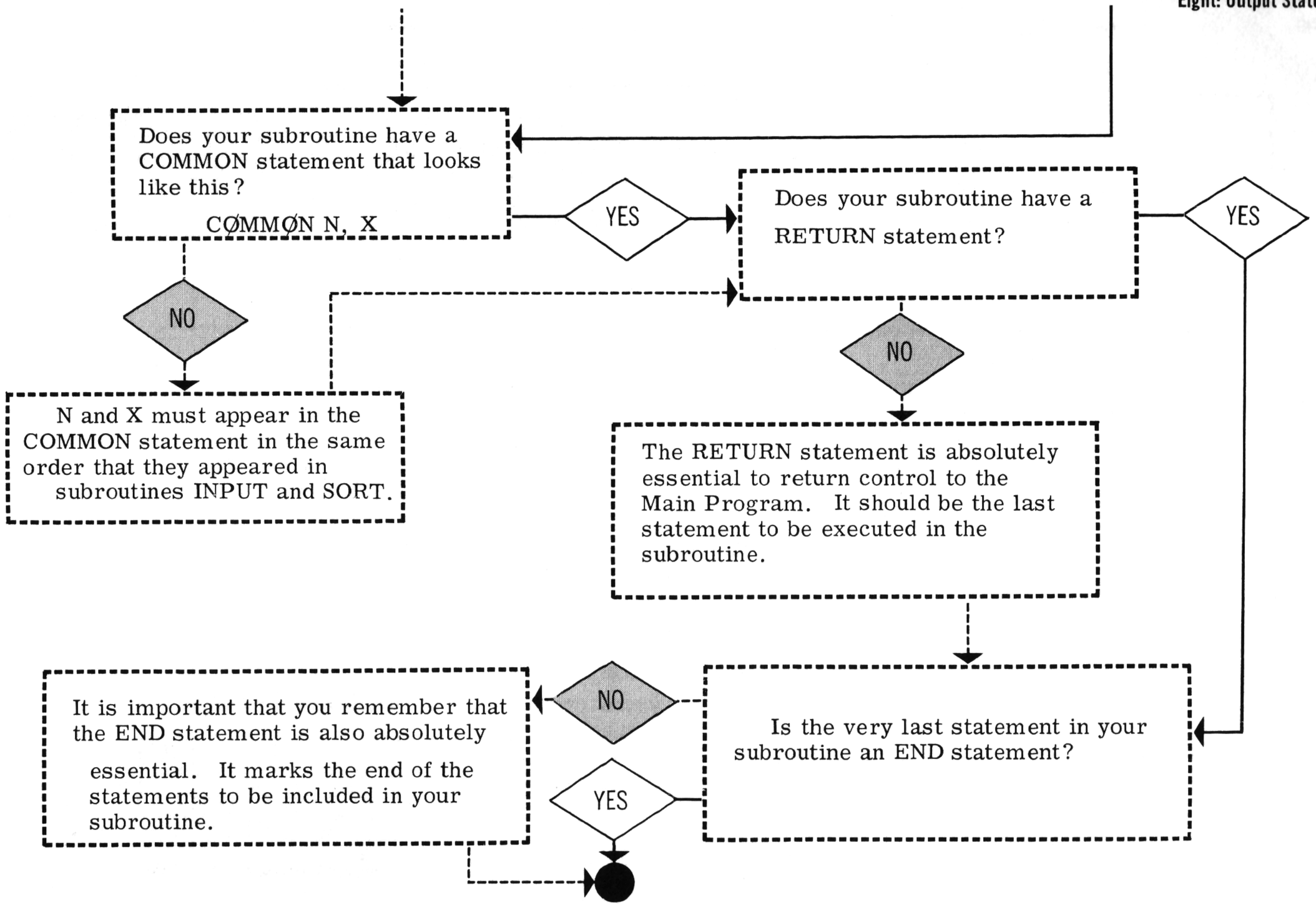
.

(The "xx.xx" notation above is used to indicate the positioning of the numbers.)

REMINDER

In some of the diagnoses, references will be made about other subroutines. These subroutines are the ones you wrote as previous TASKS.


Diagnosis
Is the first statement in your subroutine
SUBRØUTINE ØUTPUT
NO
YES
The subroutine-defining statement must be first. Otherwise, the Fortran compiler will compile this program as a Main Program.
Does your subroutine have a DIMENSION statement that looks like this?
DIMENSIØN X(1000)
NO
YES
Subscripted variables must be dimensioned in a DIMENSION statement. X is specified at 1000, because it was dimensioned that way in subroutine INPUT.


Does your subroutine have a COMMON statement that looks like this?
CØMMØN N, X
YES
NO
N and X must appear in the COMMON statement in the same order that they appeared in subroutines INPUT and SORT.
Does your subroutine have a RETURN statement?
YES
NO
The RETURN statement is absolutely essential to return control to the Main Program. It should be the last statement to be executed in the subroutine.
Is the very last statement in your subroutine an END statement?
NO
YES
It is important that you remember that the END statement is also absolutely essential. It marks the end of the statements to be included in your subroutine.

Conclusion

You have been asked to write four subprograms:

- ★ A Main Program
- ★ Subroutine INPUT
- ★ Subroutine SORT
- ★ Subroutine OUTPUT

These subprograms could be organized into a Job by keypunching each statement on tabulating cards, adding input data in a format compatible with your

INPUT Subroutine, and then adding the System Control Cards required by the computing installation where it is to be compiled and executed.

If you do have your Job run, it will probably *not* be successful the first try. You will probably get back diagnostic results from the compiler.

Every diagnostic is caused by a violation of a rule. Every rule is necessary either because of the limitations of the computer and the compiler or because it prohibits ambiguity in the program.

In one case you must serve it, and in the other case it serves you.

Answers to Exercises

15

CALL BAKER

20

Do all of them have decimal points?

If not, they *are not* Floating Point. All Floating Point Constants *must* have decimal points. Only Fixed Point constants are written with no decimal point.

25

. . . are called
Internal Variables

29

CØMMØN NAY

59

(8.0 + 8.0) / 4.0
8.0 + 8.0 is 16.0
16.0 / 4.0 is 4.0
therefore the answer is

4.0

78

variable = expression.

9

END

89

. . . statement 100, if WHICH is *negative*.
. . . statement 200, if WHICH is *zero*.
. . . statement 300, if WHICH is *positive*.
. . . always reach statement number *999*.

96

CRS(10) will be set equal to 64.0

Notice that the value of M, i.e. 64, is converted to Floating Point for storage in CRS(10)

105

```
55   FØRMAT (1HΘ,  8X,  F10.4)
```

109

25

111

```
77   FØRMAT (1H1)
     PRINT 77
```

97

9.0

because the value of Z(3) is computed the third time through the loop; i.e.
when I = 3, 3 ** 2 equals 9
9 converted to Floating Point is 9.0

100

the answer is 5 ** 2

or

25.0

1

Source

(17)

Did they all begin with one of the following characters?

I J K L M N

If not, *remember* that *all* Fixed Point variables *must* begin with one of these characters.

There are no exceptions.

Did any of them contain special characters?

(e.g. + - * / ( ) . ,)

If so, remember that special characters are *never* permitted in Fortran variables.

Did any of them have more than six characters?

Did any of them contain numbers?

(e.g. 0 1 2 3 4 5 6 7 8 9)

If yes, the use of numbers is permitted (except as the first character) but *not* recommended for beginners.

(34)

CØMMØN A, B, C

(6)

NEVER

(23)

the fifteenth

Z(15)

(24)

I = 61

(21)

Z(1)

27

```
CØMMØN   A, BIGX,  CHI
```

36

```
          1 for AJAX
        100 for BILLY
          1 for LAZY
Total   102
```

43

three

57

8.0 / 4.0 is 2.0
7.0 + 2.0 is 9.0

the answer therefore is

$\boxed{9.0}$

64

```
1.0 + 2.0 - 3.0 * 4.0 ** 2 / (5.0 + 3.0)
```

91

Control will be transferred to *statement number 30* because the value of (ABLE - BAKER) will be positive.

93

BIG will equal TABLE if TABLE was *larger* than CHECK.

104

```
      DØ  888  I = 1, N
      Y(I) = Y(I) + 10.0
888   CØNTINUE
```

102

Each time control branches to statement 54, K will be set equal to the subscript of the currently largest T. Therefore, when the loop has been satisfied, *K will be equal to the subscript of the largest value of T in the list.*

103

```
      DØ 777 I = 1, 1000
      X(I) = X(I) + 1.0
777   CØNTINUE
```

106

PRINT 55, TEST

85

. . .statement number 55.

108

One

107

PRINT 55, (Z(I), I = 1,25)

86

. . . "n" is the statement *number* of an executable statement.

88

. . . (X - XTEST) is either *zero* or *positive.*

18

Did all of them begin with one of the following characters?

ABCDEFGH ØPQRSTUVWXYZ

If not, remember that *all* Floating Point Variables *must* begin with one of the above characters.

There are no exceptions.

Did any of them contain special characters?

(e.g. + - * ( ) / . ,)

If so, remember that special characters *are not* permitted in any Fortran Variable.

Did any of them contain more than six characters?

If so, remember that six is the absolute limit.

2

Object

5

Box "J"

16

```
CALL EXIT
```

28

```
CØMMØN  DELTA,  Z,  XBAR,  JØE,  LAPD
```

30

```
CØMMØN A, B
```

3

Compilation

11

SUBRØUTINE

19

Do any of them have decimal points?

If so, they are *not* Fixed Point. Only Floating Point constants have decimal points.

31

CØMMØN A, B, C

35

300

Three hundred

58

10.0

61

(3.0 + 4.0) is 7.0

2.0 - 7.0 is -5.0

therefore the answer is

-5.0

12

A JOB

13

. . . may be a *Main* program.

62

2.0 - 3.0 is -1.0

-1.0 + 4.0 is +3.0

therefore the answer is

+3.0

66

1.0 + 2.0 - 3.0 * 16.0 / 8.0

90

If ABLE is larger than BAKER,

(ABLE - BAKER)

would be *positive.*

95

. . . will be repeated exactly 1,000 times.

69

3.0 - 6.0

48

ten

87

. . . statement number 10 if X is *negative.*
. . . statement number 20 if X is *zero.*
. . . statement number 30 if X is *positive.*

81

SUMX = SUMX + X

77

NO

Fixed Point Variables and constants may not be raised to *Floating Point* powers.

92

we may assume that the value of (GAMMA - TEST) was *positive* (because the positive (60) branch was followed) and that GAMMA was *larger* than TEST because only a smaller number (TEST) subtracted from a larger number (GAMMA) gives a positive result.

101

```
IF (BIGE - E(I)) 77,  10,  10
```

110

```
66   FØRMAT (1H0, 8F9.4)
     Print 66, (Y(I), I = 1, 80)
```

84

6.0

94

BIG will equal CHECK if CHECK was *greater* than TABLE, or if CHECK was *equal* to TABLE

99

. . . will be stored in *SUMX*.

98

the first N entries will be set equal to 1.0, because the first time through the loop Z(1) will be set equal to 1.0, the second time Z(2) will be set equal to 1.0, etc., until Z(N) is set equal to 1.0

4

INSTRUCTIONS

14

. . . the first statement is a *subroutine* or *subroutine defining* statement . . .

. . . the last statement is an *END* statement.

32

CØMMØN A, B, C

49

ten

54

16.0

22

Z(100)

26

. . . in a *Common* statement in . . .

37

specified in a *FORMAT* statement.

42

data fields

44 one

46 one hundred

60 4.0

70 -3.0

75 YES

73 FIXED POINT

56 2.0

41 three

79 DIST = TIME * RATE

80 I = I + 1

7 One hundred times

83 if A = 3.5 and B = 4.5
and X = A + B
then X = 3.5 + 4.5
therefore
X = 8.0

39

variable on the Variable List.

45

one thousand

51

B / C

55

16.0

63

A + B - C * D ** 2 / (E + C)

67

1.0 + 2.0 - 48.0 / 8.0

71

FLOATING POINT

82

SUMX = SUMX + X(I)

8

"M" times

74

FIXED POINT

33

CØMMØN A, B, C

38

is called a *data* field.

(112)

Evaluate your answer by "playing computer."

1. Assign a value to the first variable on the List in your PRINT statement.

2. Interpret the first character specified in your FORMAT statement as carriage control.

3. Print all Hollerith and "X" fields (if any) that precede the first "I" or "F" field specification.

4. If the field is an "F" field, mark off the width; print the decimal point; print the whole part of the number leaving blanks to the left of the most significant digits; then round off the fraction to the number of digits specified for the fractional part at the right, or add zeros at the right, if necessary.

5. If the field is an "I" field, mark off the width; then print the number at the extreme right of the field, leaving blanks at the left.

6. Print all remaining Hollerith and "X" fields up to the next "I" field or "F" field.

7. Assign a value to the next variable on your List and repeat steps 3, 4, 5, 6, and 7 until you come to the closing parenthesis.

8. When you come to a closing parenthesis, repeat steps 2, 3, 4, 5, 6, and 7 until all the variables have been printed.

10 RETURN

40 five

47 N

50 six

52 A + B

53 A + B

68 1.0 + 2.0 - 6.0

72 FLOATING POINT

76 FLOATING POINT

65 1.0 + 2.0 - 3.0 * 4.0 **2 / 8.0

reference manual

43

Program Structure

STATEMENTS	
44 SUBROUTINE	A Subroutine is a subprogram of the program section of a job deck. The first statement must define it as a Subroutine. The general form is: SUBRØUTINE Name "Name" may be any combination of from one to six alphabetic characters.
45 END	Every subprogram of a job deck must have an END statement as the last physical statement in the subprogram. The general form is: END

CONCEPTS		
Main Program	A subprogram is a Main Program, if the first statement in it is *not* a SUBROUTINE Statement. There must be one and only one Main Program in a Job.	47
Subprogram	A subprogram is any program segment of a Job. A Subroutine is a subprogram. A Main Program is a subprogram.	

Interprogram Flow of Control 48

STATEMENTS		
CALL	The CALL statement causes control to be transferred to the first executable statement in the subroutine named in the CALL statement. The general form is: CALL Name	47

Statements (cont.)

		"Name" may be the name of any subroutine in the Job, or any System Subroutine on the Library Tape. CALL statements may appear in Main Programs and in Subroutines.
46	RETURN	The RETURN statement in a Subroutine causes control to be transferred back to the first executable statement following the CALL statement which caused control to be transferred to the subroutine. The general form is :

```
RETURN
```

	CONCEPTS	
27	First Statement Executed	The first statement executed in a job is the first executable statement in the Main Program.
	Last Statement Executed	The last statement executed in a Job must return control to the Monitor System. One way to do this is with the following statement:

```
CALL EXIT
```

48

"EXIT" is a System Subroutine on the Library Tape. The last statement executed in a Subroutine must be either a CALL EXIT statement or a RETURN statement.

Constants

58, 63

CONCEPTS		
	Floating Point Constants	A Floating Point Constant is any number of decimal digits with decimal point. Only the first eight significant digits are used in computations, and the absolute value should lie between 10^{+38} and 10^{-38} (but may vary with different computers).

65

64 **Fixed Point Constants**

A Fixed Point Constant is one to five decimal digits with no decimal point. The absolute value must be less than 2^{17} (but may vary with different computers).

58 Variables

CONCEPTS

61 **Floating Point Variables**

A Floating Point variable is any combination of from one to six alphabetic or numeric characters and no special characters. The first character of the Floating Point Variable must be any of the following letters:

ABCDEFGH OPQRSTUVWXYZ

61 **Fixed Point Variables**

A Fixed Point variable is any combination of from one to six alphabetic or numeric characters and no special characters. The first character must be one of the following letters:

I J K L M N

66 **Subscripted Variables**

Variables of either mode may be subscripted. The general form of a single subscripted variable is:

v(i)

"v" may be any legal Floating Point or Fixed Point variable; "i" may be any legal Fixed Point variable or constant. "i" should not exceed the total number of storage locations available. For double and triple suscripted variables, see a Fortran Reference Manual.

75

Input

STATEMENTS	
82 READ	The READ statement causes values to be transmitted from data cards according to a FORMAT statement, and to be stored as the values of variables. The general form is:

READ n, v_1, v_2, v_3, . . .

"n" is the statement number of the FORMAT statement that describes the fields on the data card(s) being read.

"v_1, v_2, v_3, . . ." are the names of the variables where the values in the data fields are to be stored. These variables may be Fixed or Floating Point mode, and they may be subscripted or non-subscripted. Subscripts may be under DO-loop Control or List Control.

Input FORMAT

The input FORMAT statement contains descriptions of the data fields on the data cards to be read by a READ statement. The general form is: 77

n FØRMAT (d_1, d_2, d_3, . . .)

"n" is the statement number of the FORMAT statement.

"(d_1, d_2, d_3, . . .)" are the I Field or F Field descriptions that describe the data fields from left to right on the data

card to be read by a READ statement. Columns to the right of the data fields described in the FORMAT statement are ignored. The set of descriptions in a FORMAT statement will be repeated until values have been read for every variable on the Variable List in the Read statement. Every time the set is repeated, it applies to the next data card.

	CONCEPTS	
85	List Control	The sequence of values of the subscript of a subscripted variable may be specified in the Variable List of a READ statement by using List Control. The general form is:

$$(v(i),\ \ i=m_1, m_2, m_3)$$

"v" may be any legal variable. "i" may be any legal Fixed Point variable.

"m_1" may be any legal Fixed Point unsigned constant or variable. It is the initial value of "i." It must be greater than zero, but equal to or smaller than "m_2."

"m_2" may be any legal Fixed Point unsigned constant or variable. It is the terminal value of "i." It must be greater than zero, and equal to or larger than "m_1."

"m_3" may be any legal Fixed Point unsigned constant or variable. It is the amount by which "i" shall be incremented. It must be greater than zero. If "m_3" is not specified, it is understood to be equal to one.

Data Fields	A data field (for input) is a group of one or more columns on a punched card. The width of the data field is the number of columns in it.	75

Concepts (cont.)

78	I Fields	If the number in an input data field (on a card) is an integer number, the value should be transmitted under the I conversion symbol. The general form of the I FORMAT description is: nIw "n" is the number of fields described. If "n" is not specified, it is understood to be one. "w" is the width of the field.	
77	F Fields	If the number in an input data field (on a card) has a decimal point in it, the value should be transmitted under the F conversion symbol. The general form of the F FORMAT specification is: nFw.d "n" is the number of fields described. If "n" is not specified, it is understood to be one. "w" is the width of the field, and ".d" is the number of digits in the fractional part.	
	Repeating Input FORMAT Descriptions	If the Variable List contains more variables than there are input data field descriptions in the FORMAT statement, the FORMAT statement will be repeated from its beginning and will automatically apply to the next data card.	86
	Storage Allocation		67, 70
	STATEMENTS		
	DIMENSION	The general form of a DIMENSION statement is: DIMENSIØN $v_1(i_1)$, $v_2(i_2)$, . . .	67

		"v_1" and "v_2" . . . are the names of the variables. "i_1" and "i_2" are Fixed Point constants equal to the maximum number of locations that will ever be needed by the variable. Any number of variables may be listed in one DIMENSION statement.	**CONCEPTS**	
			Nonsubscripted Variables	Nonsubscripted variables are automatically assigned one storage location within the area used by the program in which the variable appears. If the variable is listed in a COMMON statement, the location is in Common Storage, and not in the program.
70	COMMON	The general form of a COMMON statement is:		
		CØMMØN v_1 , v_2 , . . .	Subscripted Variables	Subscripted variables are assigned as many locations within the area used by the program as specified in the DIMENSION statement in which the variable is listed. All subscripted variables must be listed in a DIMENSION statement. If the variable is also listed in a COMMON statement, the locations assigned will be in Common Storage, and not in the program. 66
		"v_1, v_2. . ." are the names of legal variables. If a variable is not subscripted, and does not appear in a DIMENSION statement, one common storage location will be allocated to it. If it is listed in a DIMENSION statement, it will be allocated as many locations as specified for it in the DIMENSION statement.	Constants	All constants are allocated one storage location within the area used by the program in which the constant appears. 63

112 **Control**

	STATEMENTS	
114	GO TO	The normal flow of control is altered by execution of a GO TO Statement. The general form to a GO TO statement is: GØ TØ n "n" is the statement number of any numbered, executable statement in the program.
115, 117	IF	The normal flow of control is altered by the execution of an IF statement. The general form of an IF statement is: IF (e) n_1, n_2, n_3 "e" is any legal expression, and "n_1, n_2, n_3," are statement numbers of any numbered, executable statements in the program. If the computed value of "e" is negative, control branches to statement number "n_1." If the computed value of "e" is zero, control branches to "n_2." If the computed value of "e" is positive, control branches to "n_3." "n_1, n_2, and n_3" need not all be different statement numbers.
120	DO	A sequence of statements may be executed repeatedly by defining the sequence as a DO-loop. The general form of a DO statement is: DØ n i $= m_1, m_2, m_3$

"n" is the statement number of the last numbered executable statement in the sequence; "i" is any legal Fixed Point Variable; and "m_1,m_2,m_3" are any legal Fixed Point variables or constants.

"i" is the index counter for the loop. "m_1" is the intial value of the index, "i." "m_2" is the terminal value of the index, "i." "m_3" is the increment for the index, "i." If "m_3" is not specified, it is understood to be one.

The "range" of a DO statement is all the statements beginning with the first one following it, and ending with statement number "n." The statements in the Range of a Do statement are executed in Normal Sequence first with $i = m_1$. After they have been executed once, i is incremented by m_3. If i does not *exceed* m_2, the statements in the range will be executed again, with i incremented. This process is repeated until the incrementing of i causes it to exceed m_2. At this time, control passes to the first statement following the range of the DO statement.

The index variable i is available for subscripting and for inclusion in expressions within the loop, but its value should never be changed by any statement in the loop. It must not, for example, appear as the index variable in an inner DO-loop or as the index variable in the list control of an input or output statement.

Control statements are permitted in a DO-loop. Control may be transferred to a statement outside the loop by an IF or a GO TO statement. Control may be transferred to a Subroutine by a CALL statement. Control may not, however, be transferred into a DO-loop except by the RETURN statement in a Subroutine called in the loop. DO-loops within DO-loops are permitted, provided only that

		no statement in the range of an inner loop lies outside the range of an outer loop. The first and last statements in the range of a DO statement must be executable. The last statement in the range of a DO statement must not be a transfer statement.	
122	CONTINUE	It is customary to mark the last statement in the range of a DO-loop with a CONTINUE statement. The general form is: n CØNTINUE "n" is the statement number of the statement. The CONTINUE statement is executable, but specifies no computation or transfer of control.	
112	Normal Flow of Control	In the normal flow of control of execution, after all the operations specified in one statement are carried out, control automatically passes to the next succeeding executable statement.	

Computations 94

CONCEPTS

Expressions	An expression is any legal Fixed Point or Floating Point variable or constant; or any legal combination of them - separated by operation symbols. When executed, an expression results in one numerical value.	94
Mixed Expressions	With only one exception, all the variables and constants in an expression must be of the same mode. The only exception is that Floating Point mode variables may be raised to Fixed Point powers.	103
Operation Symbols	The following symbols specify	94

		the following arithmetic operations: ** Exponentiation * Multiplication / Division + Addition - Subtraction	Conversion to Fixed Point	The value of a Floating Point variable may be converted to Fixed Point mode by setting the variable equal to a Fixed Point variable. When executed, the Fixed Point variable will be set equal to the whole part of the value of the Floating Point variable: The fractional part is truncated and discarded. 110
95	Hierarchy of Operations	Operations in an expression are performed according to the following preference scheme, and from left to right: 1) Exponentiation 2) Multiplication and Division 3) Addition and Subtraction	Conversion to Floating Point	Fixed Point variables may be converted to Floating Point mode by setting the variable equal to a Floating Point variable. When executed, the whole part of the value of the Floating Point variable will be set equal to the value of the Fixed Point variable, and the fractional part will be zero. 109
97	Parentheses	Those parts of an expression that are enclosed in parentheses are performed before those parts that are not enclosed. Those parts enclosed in the innermost paired parentheses are performed first; the next innermost, next.	Functions	A large number of standard functions, such as square root, sin, cosine, etc., are available. For an explanation of their use, see a Fortran Reference Manual.

147

Output

STATEMENTS	
147 PRINT	The PRINT statement causes the values of a list of variables to be transmitted for printing according to the specifications of an output FORMAT statement. The general form of a PRINT statement is: PRINT n, v_1, v_2, v_3, . . . "n" is the statement number of the output FORMAT statement specifying the layout of the printed page. "v_1, v_2, v_3, . . ." are any legal subscripted or non-subscripted variables. The value of each variable will be transmitted to the standard output tape in sequence. After the last value has been transmitted, control will pass to the next statement.
Output FORMAT	The layout of one line of output may be specified in one FORMAT statement. The general form is: 148 n FØRMAT (c, s_1, s_2, . . .) "n" is the statement number of the FORMAT statement. "c" is the carriage control field. "s_1, s_2, . . ." are output field specifications.
CONCEPTS	
List Control	The sequence of values of the subscript of a subscripted variable may be specified in the Variable List of a PRINT 155

statement by using List Control. The general form is:

$$(v(i),\ i = m_1,\ m_2,\ m_3)$$

"v" may be any legal variable. "i" may be any legal Fixed Point variable.

"m_1" may be any legal Fixed Point unsigned constant or variable. It is the initial value of "i." It must be greater than zero, but equal to or smaller than "m_2."

"m_2" may be any legal Fixed Point unsigned constant or variable. It is the terminal value of "i." It must be greater than zero, and equal to or larger than "m_1."

"m_3" may be any legal Fixed Point unsigned constant or variable. It is the amount by which "i" shall be incremented. It must be greater than zero. If "m_3" is not specified, it is understood to be equal to one.

Data Fields 149

An output data field consists of one or more print positions. The width of an output data field is the number of print positions in it.

Carriage Control 152

The following Hollerith Carriage Control fields result in the following carriage control operations:

1H1 Skip to beginning of new page
1HO Double-space before printing
1H⊙ Single space before printing
1H+ No space before printing

(Carriage Control fields may vary with different computers.)

H Fields 150

A Hollerith (H) field is a group of characters specified in a FORMAT statement, that are to be printed exactly as they

appear in the statement. The general form is:

$$nHc_1, c_2, c_3, \ldots c_n$$

"n" is the total number of characters in the field. "c_1" through "c_n," are the characters. They may be any of the alphabetic, special or numeric characters available in Fortran. There is no variable on the Variable List corresponding to an H field.

150 F Fields

Floating Point values should be printed in F-type output fields. The general form is:

nFw.d

"n" is the number of fields described. If "n" is not specified, it is understood to be one.

"w" is the "width" of the field - the number of print positions in it.

"d" is the number of decimal digits to be retained in the fractional part of the value to be printed. The fractional part of the value will be rounded to "d" digits.

If the width "w" is greater than the number of printed positions needed, the printed value will be "right-justified" within the field.

I Fields

Fixed Point values should be printed in I-type output fields. The general form is: 149

nIw

		"n" is the number of fields described. If "n" is not specified, it is understood to be one. "I" is the conversion symbol for Fixed Point values. "w" is the "width" of the field - the number of print positions in it. If the width, "w," is greater than the number of print positions needed, the value will be right-justified within the field.
150	X Fields	Fields of blanks for spacing, etc., may be specified in an output FORMAT statement. The general form is: nX "n" is the number of blanks in the field.

		"X" is the symbol for blank fields.
Repeating Output FORMAT Specifications		If the Variable List contains more variables than the output field specifications in the output FORMAT statement, the entire FORMAT statement (including carriage control) will be repeated, and will automatically apply to the next line of output.

155

Source Program Preparation

CONCEPTS	
Coding Sheets	Fortran Coding Sheets are available from computer installations, manufacturers, and business forms companies. Source Programs should be on these sheets in pencil, for easier correction. Statements should be written in large, unambiguous characters. The following conventions may be

29

followed:

0 - for zero
1 - for one
2 - for two

Ø - for the letter O
I - for the letter I
Ƶ - for the letter Z

Statements should start in column 7 and not exceed column 72. Column 6 should be used only for continuation of a previous line, and statement numbers may be written anywhere in columns 1-5.

35 **Statement Number**

Any statement may be numbered. Statement numbers, however, have no effect on the order in which the statements are executed. Numbering is for reference purposes only. Statement numbers may be written anywhere in columns 1 through 5. They must be unsigned, and should not exceed the number of storage cells in the computer. No two statements in the same program may have the same statement number.

Continuation 35

One statement may be "continued" onto another tab card (or line on a coding sheet) by writing any nonblank character in column 6. There may not be more than nine continuation cards for one statement.

Comment 57

A Comment may be inserted anywhere in any program. It is not compiled, but is listed in the printout of the Source Program. The general form is:

C Any legal characters

The character, "C" is in column one.

Data Preparation

Keypunching of data is the user's responsibility. The layout of data is extremely important. It must conform to the expectations of the program using it. It must, therefore, be carefully edited.

Index

Items in capital and lower case letters are referenced to concepts in the Reference Manual.
ITEMS IN CAPITAL LETTERS ARE REFERENCED TO STATEMENTS IN THE REFERENCE MANUAL.
Items in bold face are referenced to main headings in the Reference Manual.
Items in italics are referenced to the Program directly.

A

Accumulating 108
Arithmetic Statements 106

C

CALL 179
Carriage Control 191
Coding Sheets 193
Comment Cards 194
COMMON 185
Compilation 23
Computations 188
Constants 180
Constants, Storage of 185
Continuation 194
CONTINUE 188
Control 186
Conversion to Fixed Point 189
Conversion to Floating Point 189

D

Data Fields, Input 183
Data Preparation 194
Defining a Variable 108
DIMENSION 184
DO 186

E

END 179
Exit 180
Expressions 188

F

F Fields, Input 184
F Fields, Output 192
First Statement Executed 180
Fixed Point Constants 181
Fixed Point Variables 181
Floating Point Constants 180
Floating Point Variables 181
Flow Diagrams 6
FORMAT, Input 182
FORMAT, Output 190
Functions 189

G

GO TO 186

H

H Fields 191
Hierarchy of Operations 189

I
IF 186
I Fields, Input 184
I Fields, Output 192
Incrementing 107
Input 182

J
Job 43

L
Last Statement Executed 180
List Control, Input 183
List Control, Output 190

M
Main Program 179
Mixed Expressions 188
Multiple Fields 80

N
Nonsubscripted Variables 185
Normal Flow of Control 188

O
Object Programs 20
Operation Symbols 188
Output 190

P
Parentheses 189
PRINT 190
Program Structure 179
Punched Cards 34

R
READ 182
Repeating Input FORMAT Descriptions 184
Repeating Output FORMAT Descriptions 193
RETURN 180

S
Source Program Preparation 193
Statement Numbers 194
Storage Allocation 184
Subprogram 179
SUBROUTINE 179
Subscripted Variables 181
Subscripted Variables Storage 185

V
Variables 181

X
X Fields 193

FORTRAN CODING FORM

C ← FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ← FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ◄ FOR COMMENT	CONTINUATION	FORTRAN STATEMENT
STATEMENT NUMBER 1 5	6	7 72

FORTRAN CODING FORM

C ◄ FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ◄ FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ← FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ◄ FOR COMMENT	CONTINUATION	FORTRAN STATEMENT
STATEMENT NUMBER 1 5	6	7 72

FORTRAN CODING FORM

C ← FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ◄ FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ← FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ◄ FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ← FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ◄ FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

FORTRAN CODING FORM

C ← FOR COMMENT STATEMENT NUMBER 1 5	CONTINUATION 6	FORTRAN STATEMENT 7 72

11810